Refinement Types for ML

Tim Freeman

March 17, 1994
CMU-CS-94-110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:

Frank Pfenning, Chair
Robert Harper
Peter Lee
David MacQueen, AT&T Bell Laboratories

© 1994 Tim Freeman

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keywords: programming languages, functional programming, type inference, implicit typing, intersection types, subtyping, polymorphism

Abstract

Programming computers is a notoriously error-prone process. It is the job of the programming language designer to make this process more reliable. One approach to this is to impose some sort of typing discipline on the programs. In doing this, the programming language designer is immediately faced with a tradeoff: if the type system is too simple, it cannot accurately express important properties of the program; if it is too expressive, then mechanically checking or inferring the types becomes impractical. This thesis describes a type system called refinement types, which is an example of a new way to make this tradeoff, as well as a potentially useful system in itself.

Refinement type inference requires programs to have types in two type systems: an expressive type inference system (intersection types with subtyping) and a relatively simple type system (basic polymorphic type inference). Refinement type inference inherits some properties from each of these: as in intersection types with subtyping, we can use the type system to do abstract interpretation; as in basic polymorphic type inference, refinement type inference is decidable (preliminary experiments suggest refinement type inference may be practical as well).

We have implemented refinement type inference for a subset of Standard ML to test these ideas. We have added new syntax, called rectype declarations, to allow the programmer to specify relevant domains for the abstract interpretation. A prototype implementation of refinement type inference can do some interesting case analysis for Standard ML programs; for example, if the programmer uses a rectype declaration to declare interest in whether a boolean expression is in conjunctive normal form (CNF), refinement type inference can efficiently prove that a function for converting boolean expressions to CNF does indeed always return a boolean expression in CNF. Rectype declarations and refinement type inference seem flexible and efficient enough to practically enforce many other useful program properties as well.

Contents

1 Introduction **1**

1.1 Introductory Examples of Refinement Types 1

1.2 Practical Examples of Refinement Types 6

1.3 Related Work 10

1.4 Claims of the Thesis 12

1.5 Outline of the Work 13

2 Refinement Type Inference **15**

2.1 Introduction 15

2.2 The Formal Language 18

2.3 The Concise Language 20

2.4 Semantics 22

2.5 ML Typing 25

2.6 Monomorphic Refinement Types 29

2.7 Compatibility With ML 68

2.8 Simple Soundness Proof 80

2.9 Finite Refinements, Principality 105

2.10 Decidability 116

3 Declaring Refinements of Recursive Data Types **165**

3.1 Introduction 165

3.2 Abstract Declarations 170

3.3 Empty Types 183

3.4 Subtyping . . . 193
3.5 Splitting . . . 206
3.6 Recursive Types provide Refinement Type Constructors . . . 210

4 Refinement Type Variables 223
4.1 Adding Type Variables . . . 223
4.2 Formally Incorporating Type Variables . . . 228

5 Polymorphic Refinement Type Constructors 240
5.1 ML typing . . . 243
5.2 Subtyping . . . 243
5.3 Finiteness of Refinements . . . 246
5.4 Splitting . . . 253
5.5 Refinement Type Inference . . . 254
5.6 Soundness . . . 260
5.7 Decidability . . . 263
5.8 Declaring Polymorphic Type Constructors . . . 269

6 Declaring Refinement Types for Expressions 273

7 Implementation 275
7.1 Representations . . . 276
7.2 Refinement Type Inference . . . 281
7.3 Instantiating Refinement Types . . . 288
7.4 Analyzing Rectype Declarations . . . 297
7.5 Differences Between Implementation and Theory . . . 298

8 Conclusion, Critical Evaluation, and Future Work 300
8.1 Tradeoffs Made for Tractable Type Inference . . . 300
8.2 Experience Yet to Be Gained . . . 302
8.3 Future Work in Language Design . . . 303

8.4 Future Theoretical Work 304
8.5 Future Implementations 305

List of Figures

2.1 Monomorphic Semantics Rules 24
2.2 Monomorphic ML Typing Rules 27
2.3 Monomorphic Refinement Rules 31
2.4 Monomorphic Subtyping Rules 35
2.5 Definition of Splitting 48
2.6 Monomorphic Refinement Typing Rules 60
2.7 Decision Procedure for Refinement Types Part 1 142
2.8 Decision Procedure for Refinement Types Part 2 143

3.1 Monomorphic Recursive Type Refinement Rules 177
3.2 Whether a Value is in a Recursive Type; Greatest Fixed Point 180
3.3 When a Refinement Type is Empty 184
3.4 Declarative Emptyness for Recursive Types (Greatest Fixed Point) 186
3.5 Algorithmic Emptyness for Recursive Types 186
3.6 Declarative Rules for Recursive Subtyping (Greatest Fixed Point) 194
3.7 Algorithmic Rules for Recursive Subtyping 195
3.8 Splitting for Recursive Types (Greatest Fixed Point) 207

4.1 Sample Expression Using Polymorphism 229

5.1 Polymorphic Refinement Rules 244
5.2 Polymorphic Subtyping Rules 245

7.1 Instantiation algorithm. 296

Acknowledgements

Thanks especially to Frank Pfenning for his patience, ability to clarify half-baked ideas, careful reading of the entire thesis, and good advice in general. He also suggested `rectype` statements and lead me to believe that the problem solved by refinement types is interesting.

The remainder of my thesis committee, Peter Lee, Bob Harper, and Dave MacQueen, all provided useful advice about various aspects of the thesis. Bob Harper is especially good at rejecting nonsense; the simplicity of Chapter 6 arose when he rejected a different solution that was unnecessarily complex.

Thanks to Nevin Heintze and Benjamin Pierce for interesting and useful technical discussions. Nevin introduced me to regular tree sets, and Benjamin was an excellent example of a victorious trajectory through graduate school.

Thanks to my wife Ailing and my parents for providing some of the financial support needed to make this happen, and for being patient.

Pittsburgh, PA
January 30, 1994

Chapter 1

Introduction

In this chapter we use examples to illustrate what refinement type inference can and cannot do. We also describe the context in which this thesis exists, and give an overview of the rest of the thesis.

1.1 Introductory Examples of Refinement Types

The examples in this chapter are in Standard ML. The first example defines a Standard ML function that returns the last cons cell in a list:

```
datatype α list = nil | cons of α * α list
fun lastcons (last as cons (hd, nil)) = last
  | lastcons (cons (hd, tl)) = lastcons tl
```

Readers unfamiliar with Standard ML will benefit from some explanation of this: The first line is a `datatype` declaration that defines the ML type constructor *list* to mean LISP-like lists where all elements have the same type. The type of the elements is the argument to the type constructor; for example, since *int* is the type of integers, *int list* is the type of lists of integers. (The type is not written *list int*; unlike function application, type application is written in postfix.) This declaration also states that the constructors `cons` and `nil` can be used to construct lists.

The second and third lines are the definition of the function `lastcons`. A function definition in Standard ML consists of the keyword `fun` followed by one or more cases consisting of a function name, a pattern, an "`=`", and an expression. Each time the function is called, the first pattern that matches the actual argument is selected and used to bind variables, the corresponding expression is evaluated, and the resulting value is returned. The first pattern `(last as cons (hd, nil))` binds the variable `last` to the argument and also matches the pattern `cons (hd, nil)` against the argument; this checks that the outermost constructor is `cons` and the second argument to `cons` is `nil`. If this is so, then

we bind `hd` to the first element of the list `cons` and return `last`. The second pattern (`cons (hd, tl)`) matches any nonempty list. Since the first pattern matched lists of length one, the expression corresponding to this pattern will only be evaluated when the list has two or more elements.

The empty list `nil` is not matched by any pattern. This causes SML compilers to generate a warning during type inference that not all cases are accounted for, and an error at run time if the value `nil` is passed to `lastcons`. With refinement types, we can do better by making a declaration that distinguishes empty lists from nonempty lists. Then refinement type inference will always generate a warning when `lastcons` is used and the missing case is reachable, and it will often remain silent when the missing case of `lastcons` is unreachable. Assuming we eliminate the warning generated by SML type inference for the missing case, the net result is fewer and more specific warnings.

Standard ML also allows matching against patterns without making a function call. For example, the expression

```
case lastcons y of
   cons (x, nil) => print x
```

prints the unique element of the list returned by `lastcons`. This expression gets a compiler warning for the same reason as the definition of `lastcons`: the compiler sees that not all cases are dealt with. Once again, we can use refinement types to do better. If we make a declaration distinguishing singleton lists from other lists, refinement type inference will infer that `lastcons` always returns a singleton list and that the missing branches of this `case` statement are unreachable.

Attempting to take such refined type information into account at compile time can very quickly lead to undecidable problems. The key idea which makes our type system decidable is that subtype distinctions (such as singleton lists as a subtype of arbitrary lists) must be made explicitly by the programmer in the form of recursive type declarations. Since the programmer makes a finite number of recursive type declarations, we have a finite number of distinctions to search over during type inference.

In the example above, we can declare the refinement type of singleton lists as

```
datatype α list = nil | cons of α * α list
rectype α empty = nil
    and α singleton = cons (α, nil)
    and α long = cons (α, cons (α, α ⊤list))
    and α ⊥list = bottom (list)
```

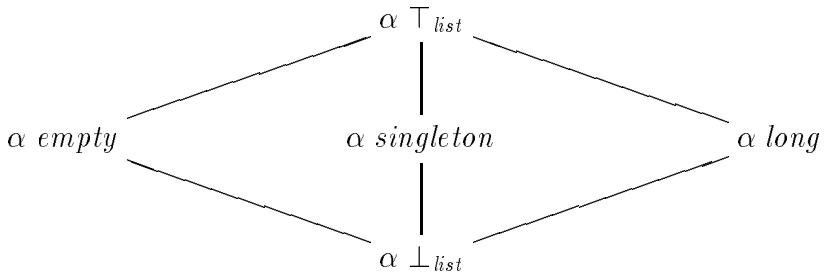
This `rectype` declaration instructs type inference to distinguish lists of length 0, 1, and 2 or more from each other. If we think of refinement types as sets, then `|` corresponds to set union. In this context, the value constructors `cons` and `nil` operate on sets; the type expression `nil` stands for the set $\{\texttt{nil}\}$ and $\texttt{cons}(X, Y)$ stands for $\{\texttt{cons}(x, y) \mid x \in X \text{ and } y \in Y\}$. The

expression `bottom` $(list)$ corresponds to the empty set of lists, and $\bot_{list}$ a refinement type identifier defined by this declaration to stand for an empty set of lists. (In the implementation, all identifiers are ASCII, so we cannot use the name $\bot_{list}$ as an identifier directly; instead we use `bot_list`. In the text of this thesis we are not limited to ASCII, so we use the name $\bot_{list}$ for this identifier.) As a convenience, the system also provides a catch-all refinement type $\top_{list}$ that includes all lists. This means that `rectype` declaration above is treated as though the clause

$$\texttt{... and } \alpha\ \top_{list} \texttt{ = nil | cons } (\alpha * \top_{list})$$

were added. (The implementation uses `top_list` instead of $\top_{list}$.)

One way to think of the refinement type inference algorithm is that it performs abstract interpretation over programmer-specified finite sets of refinement types (plural here, since each ML type has its own set of refinement types). Finiteness is important, since it is necessary for the decidability of refinement type inference. With the above declaration, abstract interpretation works over this set of refinements of $\alpha\ list$:



The system ensures that the intersection of any two refinement type constructors is also a refinement type constructor, so if we omitted the declaration of $\bot_{list}$ the lattice would look the same, except the position of $\bot_{list}$ would be occupied by an automatically generated name instead.

To perform the abstract interpretation, the type system needs to know the behaviors of `cons` and `nil` on this abstract domain. This can be expressed through refinement types given to the constructor. For example, `cons` applied to anything of type α and `nil` will return a singleton list:

$$\texttt{cons} : (\alpha * \alpha\ empty) \rightarrow \alpha\ singleton$$

The constructor `cons` also has other types, such as:

$$\begin{aligned} \texttt{cons} &: (\alpha * \alpha\ singleton) \rightarrow \alpha\ long \\ \texttt{cons} &: (\alpha * \alpha\ long) \rightarrow \alpha\ long \end{aligned}$$

In the refinement type system, we express the principal type for `cons` by using the intersection operator "$\wedge$" to combine all these types, resulting in:

$$\begin{array}{ll} \texttt{cons} : (\alpha * \alpha\ empty) & \rightarrow \alpha\ singleton \wedge \\ (\alpha * \alpha\ singleton) & \rightarrow \alpha\ long \quad \wedge \\ (\alpha * \alpha\ long) & \rightarrow \alpha\ long \end{array}$$

This type for `cons` is generated automatically from the `rectype` declaration above.

We can also use refinement types to analyze polymorphic functions. For example, we can define the usual function for applying a function to each element of a list and making a list of the results as follows:

```
fun map f nil = nil
  | map f (cons (a, b)) = cons (f a, map f b)
```

This function has the polymorphic ML type $\forall(\alpha, \beta).(\alpha \rightarrow \beta) \rightarrow \alpha\ list \rightarrow \beta\ list$. With the refinement type declarations listed above in effect, it has the refinement type

$$\begin{array}{ll} \forall(\alpha, \beta).(\alpha \rightarrow \beta) \rightarrow (\alpha\ empty & \rightarrow \beta\ empty \wedge \\ \alpha\ singleton & \rightarrow \beta\ singleton \wedge \\ \alpha\ long & \rightarrow \beta\ long). \end{array}$$

In Chapter 4, we give examples where polymorphism interferes with refinement type inference. This is not one of them; expressions using polymorphic `map` always get as precise a refinement type as similar expressions using a monomorphic version of `map`. For example, the best refinement type for `cons (nil, nil)` is $\alpha\ empty\ singleton$, and the best refinement type for `map (fn x => cons (x, nil)) (cons (nil, nil))` is $\alpha\ empty\ singleton\ singleton$. Since the value of this expression is

```
cons (cons (nil, nil), nil),
```

this is the best type we could hope for.

We can also use refinement types to prove that certain parts of a program do not use some datatype constructors. Consider a compiler for a toy language with only `if` statements, `case` statements, and variables, where `if` statements are syntactic sugar for `case` statements that operate on the booleans. It is possible to separate a compiler for this language into three sections: a parser, a desugarer that rewrites the `if` statements to `case` statements, and the rest of the compiler. Since the rest of the compiler is only given desugared code, it does not need to be prepared for `if` statements.

To formalize this, we first define the abstract syntax for this toy language:

```
datatype pat = TRUE | FALSE
datatype syn =
   VAR of string
 | IF of syn * syn * syn
 | CASE of syn * (pat * syn) list
```

We can use a `rectype` declaration to distinguish desugared abstract syntax:

```
rectype desugared =
    VAR (⊤string)
  | CASE (desugared * (⊤pat * desugared) ⊤list)
```

We omit the code for the parser. The code for the desugarer is straightforward:

```
fun desugar (IF (s1, s2, s3)) =
    CASE (desugar s1, [(TRUE, desugar s2), (FALSE, desugar s3)])
  | desugar (VAR s) = VAR s
  | desugar (CASE (s, l)) =
    CASE (desugar s, map (fn (pat, s') => (pat, desugar s')) l)
```

This gives `desugar` the refinement type $\top_{syn} \rightarrow desugared$. For the purposes of this example, our only concern about the rest of the compiler is to show that refinement type inference can verify that it does not need to deal with the `IF` constructor if it is only passed desugared code for input. It is possible to write and typecheck a caricature of the rest of the compiler by first defining a stub datatype for the output of the compiler:

```
datatype code = CODE
```

Then we define the rest of the compiler to covert all syntax it expects to encounter into a `CODE`:

```
fun rest (VAR s) = CODE
  | rest (CASE (s, l)) =
    (rest s;
     map (fn (pat, s') => rest s') l;
     CODE)
```

In this case refinement types can verify that the missing `IF` case of `rest` is never reached if its argument has the type $desugared$.

If we do not use refinement types, a dilemma arises as we write this code. Either we can have separate datatypes for the input and output of functions like `desugar`, or we can have one datatype and add unreachable cases to `rest` to keep the ML compiler from complaining. The first option is awkward because the datatypes defined tend to be redundant, and functions for printing out these datatypes (among others) must have redundant implementations. The second option is unattractive as well because the compiler does not check that the added cases are unreachable, and because we have forced the programmer to write unnecessary code.

1.2 Practical Examples of Refinement Types

Refinement type inference is practical only if, in a reasonable amount of time, it can infer useful information that is not immediately obvious to a human programmer. The examples in this section are practical in that sense; the prototype implementation needs 22 seconds elapsed time to verify them on a SPARCstation iPX and the examples are complex enough that refinement type inference found an error. This implementation is not particularly efficient; an improvement in speed by a factor of 10 would not be surprising.

We illustrate the practicality of refinement types with some code for manipulating boolean expressions. We will present this code as Standard ML syntax; it is a simple matter to translate this into the restricted language described in the theory or the language of the prototype. The assertions below about the behavior of refinement type inference are based on the behavior of the implementation; to the best of my knowledge, they are also consistent with the theory in the following chapters.

First, we can define boolean expressions with the declaration

```
datatype boolexp = And of boolexp * boolexp
                 | Or of boolexp * boolexp
                 | Not of boolexp
                 | True
                 | False
                 | Var of string
```

For example, the expression $(x \wedge y) \vee \neg(x \wedge y)$ is represented as the value

```
Or (And (Var "x", Var "y"), Not (And (Var "x", Var "y")))
```

One simple operation we can do with a boolean expression is evaluate it, if it is ground (that is, it has no variables). It is easy to write a function to do this:

```
fun eval (And (b1, b2)) = eval b1 andalso eval b2
  | eval (Or (b1, b2)) = eval b1 orelse eval b2
  | eval (Not b1) = not (eval b1)
  | eval True = true
  | eval False = false
```

Unfortunately, presenting this definition to a Standard ML system yields a warning that the function is missing a case for the `Var` constructor. This is reasonable, since we did not tell the compiler that we only intend to evaluate ground boolean expressions. With refinement types, we can tell the compiler this, and it can check that `eval` is missing no cases required to evaluate ground boolean expressions. Refinement types can also ensure at compile time that all expressions passed to `eval` are ground.

To make this happen, we define ground boolean expressions with a `rectype` declaration:

```
rectype ground = True | False | Not (ground) |
                 And (ground * ground) | Or (ground * ground)
```

This declaration defines a refinement type $ground$. Every refinement type is entirely contained within some ML type; we say the refinement type *refines* the ML type. In this case, *ground* refines *boolexp*. The `rectype` statement can be read as a description of all the ways of constructing a value with refinement type $ground$; for instance, because the `rectype` statement includes the clause `And` $(ground * ground)$, it is possible to construct a value with refinement type *ground* by applying the constructor `And` to a value with refinement type $ground * ground$; this is equivalent to saying the argument to `And` must be a pair of values, each with refinement type $ground$.

A refinement type called $\top_{boolexp}$ containing all values of ML type $boolexp$ is implicitly declared. Without this there would be no refinement type for non-ground boolean expressions, which would essentially mean that the `Var` constructor would cause a refinement type error whenever it is used.

With the declaration of *ground*, refinement type inference will infer that `eval` has the refinement type $ground \rightarrow \top_{bool}$, where $\top_{bool}$is the refinement of $bool$ that includes both `true` and `false`. As long as refinement type inference can infer that the argument passed to `eval` each time it is called has refinement type *ground*, there will be no warning and no need for a warning because the missing case in `eval` will not be reached.

Refinement types can also be used to infer useful things about the result of substituting values for variables in boolean expressions. This requires manipulating substitutions; we will represent a substitution as a value with the type $(string * boolexp)\ list$,where the first element of each pair in the list is the name of the variable and the second is the corresponding value. An elementary operation on substitutions is looking up a value in a substitution, which can be implemented with the code:

```
fun lookup (cons ((st1, v), tl)) st2 =
    if st1 = st2 then v else lookup tl st2
  | lookup [] _ = error ()
```

If we assume `error` has the ML type $\alpha \rightarrow \beta$, then `lookup` gets the ML type $(\underline{\alpha} * \beta)\ list \rightarrow \underline{\alpha} \rightarrow \beta$, where we underline type variables for which polymorphic equality must be defined. The refinement type inferred for it is similar: $(\underline{\alpha} * \beta)\ \top_{list} \rightarrow \underline{\alpha} \rightarrow \beta$.

When we instantiate the ML type variables $\underline{\alpha}$ and β, the corresponding refinement type variables can be instantiated to any refinement of the ML type substituted for the corresponding ML type variable. For example, consider the instantiation mapping $\underline{\alpha}$ to *string* and β to *boolexp*. Instantiating the ML type $(\underline{\alpha} * \beta)\ list \rightarrow \underline{\alpha} \rightarrow \beta$ yields the ML type $(string * boolexp)\ list \rightarrow string \rightarrow boolexp$. We will suppose that $\top_{string}$ refines *string*; since $\top_{boolexp}$ refines $boolexp$, instantiating the refinement type $(\underline{\alpha} * \beta)\ \top_{list} \rightarrow \underline{\alpha} \rightarrow \beta$ yields

$$((\top_{string} * \top_{boolexp})\ list \rightarrow \top_{string} \rightarrow \top_{boolexp}\,.$$

Since $ground$ also refines $boolexp$, instantiating it also yields

$$(\top_{string} * ground)\ list \rightarrow \top_{string} \rightarrow ground.$$

We can combine multiple refinement types of an expression with $\wedge$, so `lookup` also has the refinement type

$$(\top_{string} * ground)\ list \rightarrow \top_{string} \rightarrow ground \wedge (\top_{string} * \top_{boolexp})\ list \rightarrow \top_{string} \rightarrow \top_{boolexp}.$$

Now we can use `lookup` to implement a function for applying a substitution:

```
fun apsubst (And (b1, b2)) s =
      And (apsubst b1 s, apsubst b2 s)
  | apsubst (Or (b1, b2)) s =
      Or (apsubst b1 s, apsubst b2 s)
  | apsubst (Not b1) s = Not (apsubst b1 s)
  | apsubst (Var st) s = lookup s st
  | apsubst x _ = x
```

and with refinement types we can infer that this function has the type

$$\top_{boolexp} \rightarrow (\top_{string} * ground)\ \top_{list} \rightarrow ground,$$

which means that applying a ground substitution to any boolean expression yields a ground boolean expression (or raises an exception).

We can also use refinement types to reason about boolean expressions in conjunctive normal form (CNF). We can distinguish these with the following `rectype` declaration:

```
rectype cnf = And (cnf * cnf) | disj | True
    and disj = Or (disj * disj) | literal | False
    and literal = Not (atom) | atom
    and atom = Var (⊤string)
```

As an example of the use of this `rectype` statement, we can write a function to convert boolean expressions into CNF, and use refinement types to verify that it always returns an expression in CNF. We define the function in two steps; the first step is to transform the disjunction of two expressions in CNF into an expression in CNF:

```
fun disjCnfs (And (b1, b2)) c = And (disjCnfs b1 c, disjCnfs b2 c)
  | disjCnfs True c = True
  | disjCnfs b (And (c1, c2)) =
      And (disjCnfs b c1, disjCnfs b c2)
  | disjCnfs b True = True
  | disjCnfs b c = Or (b, c)
```

Type inference infers that `disjCnfs` has the refinement type $cnf \rightarrow cnf \rightarrow cnf$, among others.

An earlier version of this had an error that was found when the earlier definition did not have the correct refinement type. The version with the error was

```
fun disjCnfs (And (b1, b2)) c = And (disjCnfs b1 c, disjCnfs b2 c)
  | disjCnfs True c = True
  | disjCnfs (b as Or _) (And (c1, c2)) =
      And (disjCnfs b c1, disjCnfs b c2)
  | disjCnfs (b as Or _) True = True
  | disjCnfs b c = Or (b, c)
```

An example of the error is `disjCnfs False (And (False, False))`; this evaluates to

```
Or (False, And (False, False)),
```

which is not in CNF, even though both of the arguments to `disjCnfs` are in CNF. The prototype implementation detected the error when it was told to check the assertion that `disjCnfs` has the type $cnf \rightarrow cnf \rightarrow cnf$; the command to do this is written as

```
val _ = disjCnfs ◁ cnf → cnf → cnf
```

(Actually, the prototype implementation only takes ASCII characters for input, so it is really written

```
val _ = disjCnfs <: cnf -> cnf -> cnf
```

However, since the implementation is a research prototype rather than a practical tool at this point, readability is more important than gritty realism, so we will typeset all discussion of the implementation.)

After we can convert the disjunction of two CNF boolean expressions to CNF, it is easy to convert arbitrary boolean expressions to CNF:

```
fun toCnf (And (b1, b2)) = And (toCnf b1, toCnf b2)
  | toCnf (Or (b1, b2)) = disjCnfs (toCnf b1) (toCnf b2)
  | toCnf (Not (And (b1, b2))) = toCnf (Or (Not b1, Not b2))
  | toCnf (Not (Or (b1, b2))) = toCnf (And (Not b1, Not b2))
  | toCnf (Not True) = False
  | toCnf (Not False) = True
  | toCnf (Not (Var s)) = Not (Var s)
  | toCnf z = z
```

The prototype implementation can infer that `toCnf` has the refinement type $\top_{boolexp} \rightarrow cnf$.

1.3 Related Work

The main features of refinement type inference – basic polymorphic type inference, subtyping, intersection types, and `rectype` declarations – are all derived from features of languages that have appeared in the literature.

1.3.1 Basic Polymorphic Type Inference

A dynamically typed language like LISP can have one function that can append any kinds of lists, whether those lists contain integers, booleans, or other lists. Parametric polymorphism provides some of this flexibility to statically typed languages. In this example, we start by assuming all elements of the lists have the same type; we will name this type with a parameter, say α. Then, for all α, the function that appends lists (call it `append`) can have the type $\alpha\ list \rightarrow \alpha\ list \rightarrow \alpha\ list$. We can see that this function can append lists of booleans by first *instantiating* α to $bool$, to conclude that `append` also has the type $bool\ list \rightarrow bool\ list \rightarrow bool\ list$.

Standard ML is a language with parametric polymorphism that has been developed for at least 15 years [Mil78, MTH90, MT91b, Har86, HMM+88, Tof87, Tof88, DM82, Mac88]; practical implementations are freely available. Standard ML has the added advantage that the polymorphism is *implicit*, which means that no types need be mentioned in the definition of functions such as `append`; instead, they can all be inferred.

Refinement type inference simply uses polymorphic type inference with minimal changes. Ultimately, we hope to have a dialect of Standard ML that will accept all existing SML programs, as well as SML programs with `rectype` declarations added. Since a large body of SML code already exists, this may lead to widespread use of refinement types fairly soon after good implementations of refinement types are available.

1.3.2 Subtyping

Roughly speaking, one type is a *subtype* of another if all values with the first type also have the second type. For example, in mathematics, all integers are real numbers, so programming languages often have the type of integers (which we shall call int) as a subtype of the type of real numbers (which we shall call $real$). Since integers are often implemented differently from real numbers, the simple notion of subtyping as containment is not necessarily true at the implementation level; instead, we may have to use some non-trivial function to coerce elements of the subtype into elements of the supertype. In this example, the coercion function maps the machine representations of integers into the machine representations of reals.

Subtyping at a base type leads naturally to subtyping at higher types; for example, if int is a subtype of $real$, then we would expect $(int * int)$ to be a subtype of $(real * real)$. We

also have the slightly counter-intuitive assertion that $real \to bool$ is a subtype of $int \to bool$; this is the case because any element of the former can be converted to an element of the latter by first coercing the argument of the function from int to $real$.

Subtyping also has a natural interpretation when used with records. Any record with, say, both `age` and `name` fields has an `age` field; if we rephrase this in terms of subtyping and standard notation for record types, and assuming that the `age` field is an int and the `name` field is a $string$, we say that $\{\texttt{age} : int, \texttt{name} : string\}$ is a subtype of $\{\texttt{age} : int\}$. Several approaches to clean interaction between this kind of subtyping and polymorphism are [JM88, Jat89, R89́, LW91, HP91, HL94]. These are all type inference systems that in some sense extend the type inference of Standard ML; making a version of refinement types that is based on one of these instead of Standard ML is potential future work.

Two papers by Fuh and Mishra [FM89, FM90] describe an interesting system that deals simultaneously with polymorphism and subtyping. Like refinement types, their system permits a user-defined subtyping relation, but unlike refinement types their system has no intersection operator. In their system the result of type inference is a pair, consisting of a type and a set of constraints that may stipulate that some free type variables appearing in the type must be subtypes of one another. It is not clear how to extend this system to include intersections.

Subtyping in refinement types is simpler than subtyping in general because with refinement types, the coercion function is always the identity function.

1.3.3 Intersection Types

Intersection types record multiple pieces of information about an expression. For example, consider a unary negation operator that applies to both integers and reals. It will therefore have both of the types $int \to int$ and $real \to real$. Therefore, if we have intersection types in our language, we can say it has the type $(int \to int) \wedge (real \to real)$. When the type system uses the type of unary negation, it will have to select an appropriate type from the ones intersected.

These types can quickly become too expressive; type inference becomes undecidable [CDCV80]. There are restrictions that yield a decidable system [CG92], but it is not clear what subtyping means in this system.

Another variant of this is Forsythe [Rey88]. This language has intersection types and subtyping, but no polymorphism. This language has a different approach to records from the polymorphic one mentioned above: the intersection of the two record types $\{\texttt{age} : int\}$ and $\{\texttt{name} : string\}$ is $\{\texttt{age} : int, \texttt{name} : string\}$.

Yet another option is $F_\wedge$ [Pie91b]. This language has intersection types, subtyping, and polymorphism, and its type system can encode any of the abstract interpretations that refinement type inference can. However, it has explicit types, and type checking in this system is undecidable.

1.3.4 `Rectype` Declarations

`Rectype` declarations essentially define finite automata that recognize when a value is in a refinement type. In fact, as long as there are no function types, a `rectype` declaration has exactly the same descriptive power as a regular tree automaton [GS84]. Similar automata have been used to define types for logic programs; most of the papers in [Pfe92] deal with some aspect of this. For example, [YFS92, page 68] uses the example

$$Evenlist(\tau_1, \tau_2) ::= [\,]; [\tau_1 \mid Oddlist(\tau_1, \tau_2)].$$
$$Oddlist(\tau_1, \tau_2) ::= [\tau_2 \mid Evenlist(\tau_1, \tau_2)].$$

which is remarkably similar to (and probably derived independently from) the example we will use in the next chapter:

$$\texttt{datatype } blist \texttt{ = nil | cons of } bool * blist$$
$$\texttt{rectype } bev \texttt{ = cons } (\top_{bool} * bod) \texttt{ | nil } (\textit{runit})$$
$$\texttt{ and } bod \texttt{ = cons } (\top_{bool} * bev)$$

The operations needed to determine the meaning of a `rectype` declaration can be computed exactly for regular tree automata [GS84]. The algorithms given for regular tree automata in [DZ92] seem practical, and assuming they are sound, they are more accurate than the type system given in Chapter 3. An example where the current specification is weak is on page 193. Finding a practical algorithm that works well in the general case and is exact in the case when there are no function objects is future work.

1.4 Claims of the Thesis

The central claim of this thesis is:

> Refinement types provide a sound, practical, declarative, and unobtrusive way to express and effectively verify some reasoning by cases about Standard ML programs and potentially programs in other functional programming languages.

This has several parts:

- Refinement types can express reasoning by cases about Standard ML programs. This requires subtyping (because some cases include others) and intersection types (to describe the behavior of functions on multiple possible inputs).
- Refinement types can be effectively verified. This means that type inference is decidable; we ensure this by only distinguishing cases in which the programmer has declared interest and by requiring expressions with a refinement type to always have an ML type.

- Refinement type inference is practical. This thesis demonstrates this by describing a prototype implementation of it that has tolerable performance.
- Refinement type inference is sound. We exhibit a soundness proof.
- Refinement type inference is declarative. Refinement type inference can be described with a declarative type inference system. See Figure 2.6 on page 60.
- Refinement type inference is unobtrusive. If `rectype` declarations and "$\lhd$" are not used in a program, then that program has a refinement type if and only if it has an ML type. Refinement types are also unobtrusive in the sense that they do not necessarily have any effect on execution. Given a program that checks refinement types, it is easy to construct a compiler for a variant of Standard ML that has refinement types: the compiler could simply check refinement types, remove all `rectype` declarations and uses of "$\lhd$" from the given source code, and then pass the resulting source code to a Standard ML compiler.

1.5 Outline of the Work

This thesis starts with a careful formal description of monomorphic refinement type inference. Chapter 2 centers around the inference rules in Figure 2.6 that describe refinement type inference for expressions in terms of explicit assumptions about properties of the information from `rectype` statements. The rest of that chapter consists of proofs that this type inference system is sound, has principal types, and is decidable.

Chapter 3 deals with `rectype` statements. The central inference systems are Figure 3.6, which describes how to infer a subtyping relation from a `rectype` declaration, and Figure 3.8, which describes how to infer the splitting relation. The rest of the chapter consists of proofs that the assumptions made in Chapter 2 are satisfied, and a proof that refinement type inference for values is consistent with a semantics we give for `rectype` statements.

Chapters 4 and 5 describe how to add polymorphism to this. Chapter 4 simply adds type variables such as α, and is fairly simple. Chapter 5 add type constructors that take type arguments, such as $\alpha\ list$. This is more complex because we have to decide whether, for example, $(bev \wedge bod)\ singleton \leq bev\ singleton$. There are four possible ways a refinement type constructor can change when we replace its argument by a larger argument: either it gets larger, gets smaller, stays the same, or the new type is incomparable with the old. We call these type arguments positive, negative, ignored, and mixed, respectively. It is possible to create examples of all of these behaviors, and the theory has to deal with them. Chapter 5 describes the changes necessary to the reasoning in Chapters 2 and 3 to accommodate this.

Chapter 6 describes how to add the coercion operator $\lhd$ to the language. This is very straightforward, provided one erases all coercion operators from terms before evaluating them.

Chapter 7 describes the prototype implementation. The fundamental decision made in the implementation was to implicitly represent refinements of functional ML types as functions. Memoization is used extensively. We find fixed points by using pending analysis [Jag89, Dix88], and we instantiate polymorphic refinement types using a novel unproven strategy that appears to yield correct answers in practice.

Chapter 2

Refinement Type Inference

2.1 Introduction

This chapter gives a formal description of a simple, monomorphic form of refinement types that includes only primitives for functional programming and assumes that the programmer has already declared which distinctions he is interested in. In Chapter 3, we describe `rectype` statements, which allow the programmer to declare interest in specific distinctions. The soundness results of this chapter depend on several assertions that are proved in Chapter 3. These assertions are labeled as "Assumptions"; for example, the first one below is Assumption 2.2 (Constructors have Unique ML Types) on page 26. Chapter 4 expands the type inference in this chapter to include type variables, and then Chapter 5 adds polymorphic constructors.

Perhaps the simplest example of refinement types is being able to reason about the booleans. If the programmer has declared interest in the distinction between `true` and `false`, refinement type inference can determine that the function

$$\texttt{fn x => or (x, not x)}$$

always returns `true` regardless of its input. To express this formally, we say

$$\vdash \texttt{fn x => or (x, not x)} : \top_{bool} \rightarrow tt.$$

Here $\top_{bool}$, tt, and $\top_{bool} \rightarrow tt$ are all refinement types. Informally we can think of refinement types as standing for sets of values. The refinement type $\top_{bool}$ is the set of all boolean values, or $\{\texttt{true}, \texttt{false}\}$; tt is the set $\{\texttt{true}\}$; and $\top_{bool} \rightarrow tt$ is the set of all functions with ML type $bool \rightarrow bool$ that map all values in $\top_{bool}$ to values in tt.

If we think of refinement types as sets of values, each value in the set must have the same ML type; we say that the refinement type refines the ML type. For example, $\top_{bool}$ refines $bool$ and $\top_{bool} \rightarrow tt$ refines $bool \rightarrow bool$. For the purposes of the examples in this

chapter, the refinements of $bool$ are

$\top_{bool}$, corresponding to the set $\{\texttt{true}, \texttt{false}\}$
tt, corresponding to the set $\{\texttt{true}\}$
ff, corresponding to the set $\{\texttt{false}\}$
$\bot_{bool}$, corresponding to the set $\{\}$

where the symbol $\bot_{bool}$ is a typeset version of the name `bot_bool`. The symbols "$\bot$" and "$\top$" by themselves have no meaning in this thesis.

This example has some special features that will not hold in general. Although there is always a least refinement of every ML type, in general there may be values of that refinement type, unlike this example where there are no values of type $\bot_{bool}$. The simplest instance of this arises when the programmer has not asked for any refinement type distinctions; when this happens, there is exactly one refinement of each ML type, and there are always values with that refinement type. Since we want the programmer to have the option of ignoring refinement types, and having multiple refinements of an ML type slows down type inference, we should only have multiple refinements of an ML type when the programmer asks for it.

In this example, there is a maximal refinement $\top_{bool}$. In Chapter 5 we will mention examples involving polymorphism where there is no maximal refinement type.

In Forsythe [Rey88], there is a maximal type called "ns". Every Forsythe expression has this type, possibly among others; when the type system detects that an expression is ill-behaved, it has only the type ns. We do not take this approach. Instead, the refinement type system takes the more conventional approach of ensuring that ill-behaved expressions have no type. This begins to be inconvenient in Section 2.9, where to simplify the statement of some theorems we introduce the notion of "generalized refinement types", each of which is either a refinement type or ns. Even then, ns is not the refinement type of any expression.

The meaning of ns is entirely different from the meaning of $\top_{bool}$. There are perfectly well-behaved expressions with the refinement type $\top_{bool}$; however, ns is not a refinement type, and it is not used to describe the behavior of well-behaved expressions.

According to the above list of the refinements of $bool$, the intersection of two refinements of $bool$ is a refinement of $bool$. This is a desirable property, but we need to add an operator to make it continue to hold for refinements of other ML types; for example, with the notation introduced so far, we cannot write a refinement type that is the intersection of $tt \to ff$ and $ff \to tt$. We will call this operator "$\wedge$". For example, here are some of the refinements of $bool \to bool$ and some of the elements of the set corresponding to each one:

$\top_{bool} \to tt$ contains the elements `fn x => true`
and `fn x => or (x, not x)`
$tt \to ff \wedge ff \to tt$ contains the elements `not`
and `fn x => or (not x, not x)`
$\top_{bool} \to \top_{bool}$ contains all values with ML type $bool \to bool$
$tt \to \top_{bool} \wedge \top_{bool} \to tt$ is equivalent to $\top_{bool} \to tt$

Two refinement types are equivalent if they correspond to sets with the same elements, otherwise they are distinct. For instance, the refinement type $tt \wedge ff \wedge tt$ is equivalent to $\bot_{bool}$. A largest set of distinct refinements of $bool$ is $\{\top_{bool}, tt, ff, \bot_{bool}\}$, so $bool$ has only finitely many distinct refinements. It turns out that all ML types only have finitely many distinct refinements; this is true for datatypes because the programmer only has time to specify a finite number of distinctions, and it is true for other ML types because the operators "$\rightarrow$" and "$*$" we use to construct ML types from other ML types do not introduce infinite numbers of distinct refinements where there were none before. This is the crucial property that makes refinement type inference decidable; we prove it in Section 2.9.

Refinement types become more interesting and useful when we use them with recursive datatypes. A simple example of this is representing nonnegative integers as strings of bits:

```
datatype bitstr = Zero of bitstr | One of bitstr | Empty
```

Suppose the least significant bits are outermost, so that `Zero (One Empty)` represents the integer 2. Every nonnegative integer has multiple representations in this system; for example, another representation of 2 is `Zero (One (Zero Empty))`. Every positive integer, however, does have exactly one representation that does not have `Zero` as the most significant bit; call this "normal form". We can define a refinement type nf containing just the positive integers in normal form, and we can prove that straightforward functions for doing arithmetic such as

```
fun add (One b1) (One b2) = Zero (add (add (One Empty) b1) b2)
  | add (One b1) (Zero b2) = One (add b1 b2)
  | add (Zero b1) (One b2) = One (add b1 b2)
  | add (Zero b1) (Zero b2) = Zero (add b1 b2)
  | add Empty b = b
  | add b Empty = b;
```

return values of type nf when passed values of type nf.

Refinement type inference determines the type of an expression by first finding types for the subexpressions. Thus, we can only discover that `One Empty` has the type nf if we first have a type for `Empty`. We will call that type em. We shall assume that $bitstr$ has the following refinements:

$\bot_{bitstr}$ corresponds to the empty set
em corresponds to $\{\text{Empty}\}$
nf corresponds to positive integers that are in normal form
$\top_{bitstr}$ corresponds to the set of all bitstrings

In this case the set of values of type *nf* is clearly infinite. This makes it clear that any implementation must use some representation of refinement types other than sets of values. In both the implementation and the formal description of refinement types, we assign

refinement types to constructors instead of representing refinement types as sets of values. Since, in this formalism, values are expressions, we can use refinement type inference to determine which values are in which refinement types; for example, the assertion

$$\texttt{Zero (One Empty)} \in nf$$

becomes

$$\vdash \texttt{Zero (One Empty)} : nf.$$

In this chapter, we start by formally defining our object language in Section 2.2. We also describe an alternative, more concise syntax in Section 2.3. A formal semantics is in Section 2.4. To provide background for the description of refinement types, we define a restricted version of the usual ML type system in Section 2.5. A simple version of refinement types is defined in Section 2.6. We prove that it is compatible with ML type inference in Section 2.7, and sound in Section 2.8. We show that each ML type has finitely many refinement types in Section 2.9, and use this fact to give a decision procedure for refinement types in Section 2.10.

2.2 The Formal Language

Each language we define in this thesis will have two kinds of types. The more familiar kind resembles the one commonly used for SML; we shall call these ML types. In later sections of this chapter we will be defining more informative types with an intersection operator "$\wedge$"; we shall call these refinement types.

We shall use the metavariables t and u to stand for ML types throughout this thesis. The metavariable tc stands for an ML type constructor, so we can define the language of ML types with the grammar

$$t ::= tc \mid t * \ldots * t \mid tunit \mid t \rightarrow t.$$

In SML the type of a zero-way tuple is called $unit$. Here we call it $tunit$ instead to distinguish the ML type for empty tuples from the refinement type for empty tuples that we will introduce later.

We could slightly simplify the presentation in this chapter by replacing the arbitrary-length tuple types here with binary and nullary tuples. However, when we introduce polymorphic constructors in Chapter 5, tuples will become a polymorphic data type very similar to other polymorphic data types, and at that point arbitrary length tuples will add little to the complexity of the theory. Thus we will use arbitrary length tuples here to simplify the analogy between the system described in this chapter and the system described in Chapter 5.

We use x, y, and f as metavariables to stand for object language variables, c to stand for constructors, and e to stand for expressions. The metavariables x, y, and f that appear

often in the mathematics should not be confused with the object language variables `x`, `y`, and `f` that appear often in the examples. Expressions have the following grammar:

$$
\begin{aligned}
e ::= {} & x \mid \texttt{fn}\ x{:}t\ \texttt{=>}\ e \mid e\ e \mid c\ e \mid \\
& \texttt{case}\ e\ \texttt{of}\ c\ \texttt{=>}\ e\ \texttt{|}\ \ldots\ \texttt{|}\ c\ \texttt{=>}\ e\ \texttt{end}{:}t \mid \\
& (e, \ldots, e) \mid () \mid \texttt{elt\_}m\texttt{\_}n\ e \mid \\
& \texttt{fix}\ f{:}t\ \texttt{=>}\ \texttt{fn}\ x{:}t\ \texttt{=>}\ e
\end{aligned}
$$

Notice that this grammer uses the "|" operator as meta-syntax to describe a language containing the character `|` in the syntax of `case` statements. This language is roughly a monomorphic version of Mini-ML [CDDK86].

As the grammar says, all constructors take exactly one argument, which may be a tuple. We use $()$ to mean a tuple of zero elements. Common constructors include `true` and `false`; thus `true` is not a syntactically valid expression in itself, but `true` $()$ is.

There are explicit types appearing at several places in the grammar. The ML types after each variable binding in abstractions and fixed points ensure that each expression has at most one ML type derivation; the need for this is discussed in the next section. The ML type at the close of each `case` statement prevents obscure pathological behavior that would prevent Theorem 2.54 (Inferred Types Refine) on page 68 from being true; see page 68 for a discussion.

As in Standard ML [MTH90, MT91b], the fixed point operator can only apply to functions. This outlaws oddities such as `fix f => not f`. It is possible that the theory below could be adjusted to permit recursive values, but they seem troublesome and not particularly useful, so we shall avoid them.

2.2.1 Explicit or Implicit ML Types

One of the major features of SML is that it is implicitly typed. This frees the programmer from most of the burden of type declarations. Since our goal is to analyze Standard ML, the language our implementation starts with must also be implicitly typed. However, since ML type inference is well understood, we have the option of assuming that the expressions analyzed by the refinement type system described here have already had explicit types inserted by ML type inference. The purpose of this section is to explain why we take this option.

The problem with implicit ML types is that there are sometimes multiple derivations of an ML type for an expression. For example, consider the SML declaration

```
val foo = (fn y => (fn z => z)) (fn x => x)
```

and suppose `foo` has the ML type $bool \to bool$. Even though we know the type of `foo`, there are still many different ways to derive this type because we can give the subexpression

`fn x => x` the ML type $bool \to bool$ or $(bool * bool) \to (bool * bool)$ or, in general, $t \to t$ for any ML type t. Once we add polymorphic type variables to the system in Chapter 4, we will be able to give examples where different ML type derivations lead to different refinement types for the expression.

However, it seems that without polymorphism, the refinement type assigned to an expression depends only on the ML type assigned to the expression, not on how that type is derived. For instance, `foo` has the ML type $t \to t$ for any t. If we choose an ML type for `foo` by choosing t to be $bool$, then the expression has a principal refinement type

$$tt \to tt \wedge ff \to ff \wedge \top_{bool} \to \top_{bool} \wedge \bot_{bool} \to \bot_{bool}$$

that does not depend on which ML type we assign to the `fn x => x` subexpression. It seems that the ML type of an expression uniquely determines its principal refinement type in general, since if the `fn x => x` subexpression were used, its ML type would be constrained.

Since this property will not continue to hold when we add polymorphism, it seems better to add explicit types to the terms now to ensure a unique ML type derivation than to prove that knowing the ML type is sufficient in the special case of monomorphic expressions. To ensure a unique ML type derivation, we write the ML type for each bound variable. For example, the two derivations mentioned above of an ML type for `foo` correspond to these translations of the definition of `foo` into monomorphic expressions:

$$\begin{array}{l}(\texttt{fn y}:bool \to bool \texttt{ => } (\texttt{fn z}:bool \texttt{ => z})) \\ \quad (\texttt{fn x}:bool \texttt{ => x})\end{array}$$

and

$$\begin{array}{l}(\texttt{fn y}:(bool * bool) \to (bool * bool) \texttt{ => } (\texttt{fn z}:bool \texttt{ => z})) \\ \quad (\texttt{fn x}:bool * bool \texttt{ => x}).\end{array}$$

2.3 The Concise Language

For brevity, we will want to have implicit types in our examples. Thus we shall also have an informal, concise syntax where we omit the types with the understanding that the real expression has some consistent ML types inserted. This notation is only unambiguous when the concise expression has a unique type derivation. Roughly speaking, our concise language is the subset of SML that can be easily translated to fit the grammar for expressions on page 19. We will have the following differences between the concise language and the formal language:

- The concise language has constant value constructors, but in the formal language all constructors take one argument. Eliminating constant value constructors from the formal language decreases the number of cases that have to be considered in the

proofs. Since we can always encode a constant value constructor as a function one that takes an argument of ML type $tunit$, this does not decrease the expressiveness of the language.

- The concise language uses destructuring in `fn` expressions to extract an element from a tuple, but the formal language uses the `elt_`m`_`n primitive to extract the mth element from a tuple of n elements. Eliminating destructuring from `fn` expressions simplifies the proofs. Encoding the length of the tuple in the operator for extracting elements eliminates some ambiguity; for example, in SML the expression `#1` by itself is not valid because it is not clear whether to give it the type $\alpha * \beta \rightarrow \alpha$ or the type $\alpha * \beta * \gamma \rightarrow \alpha$ or one of the infinitely many other possibilities. In the formal language, this sort of ambiguity does not arise.

- The concise language has `case` statements that bind variables, but the formal language does not; in the formal language, the only constructs that bind a value to a variable are abstractions and fixed points. For instance, if we assume that lists of booleans have been defined with the datatype

$$\texttt{datatype}\ blist\ \texttt{=}\ \texttt{nil}\ \texttt{|}\ \texttt{cons of}\ bool * blist$$

  the concise expression

```
case cons (true, nil) of
   cons (x, y) => y
 | nil => nil
```

  corresponds to the formal expression

```
case cons (true (), nil ()) of
   cons => fn p:bool * blist => elt_2_2 p
 | nil => fn ignored:tunit => nil ()
end:blist
```

- The concise language has only enough type declarations to uniquely determine the type derivation, whereas the formal language has type declarations throughout the code.

- The concise language has `let` statements, but the formal language does not. Since the formal language does not have polymorphism, each statement of the form

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}$$

  can be interpreted as the expression

$$(\texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ e_2)\ e_1$$

in the formal language, for some appropriate t.

- The concise language freely uses many of SML's convenient syntactic features that are omitted from the formal language, such as let statements and abstraction operators that take cases, destructure tuples, and define curried functions.

For example, the concise language expression

```
let fun double f x = f (f x)
    fun not true = false
      | not false = true
in
    double not true
end
```

corresponds to exactly one formal expression:

$$
\begin{array}{l}
(\texttt{fn double}:(\mathit{bool} \to \mathit{bool}) \to \mathit{bool} \to \mathit{bool}\ \texttt{=>} \\
\quad ((\texttt{fn not}:\mathit{bool} \to \mathit{bool}\ \texttt{=>} \\
\qquad \texttt{double not (true ())}) \\
\quad (\texttt{fn b}:\mathit{bool}\ \texttt{=>} \\
\qquad \texttt{case b of} \\
\qquad\quad \texttt{true => fn ignored}:\mathit{tunit}\ \texttt{=> false ()} \\
\qquad\ \ \texttt{| false => fn ignored}:\mathit{tunit}\ \texttt{=> true ()} \\
\qquad \texttt{end}:\mathit{bool})) \\
(\texttt{fn f}:\mathit{bool} \to \mathit{bool}\ \texttt{=> fn x}:\mathit{bool}\ \texttt{=> f (f x)}))
\end{array}
$$

2.4 Semantics

This section describes how to evaluate closed expressions. We will call the result of evaluation a value; every value is a closed expression of the form

$$v ::= c\ \ v \mid (v, \ldots, v) \mid () \mid \texttt{fn}\ x{:}t\ \texttt{=>}\ e.$$

Since our values are expressions, we can apply the same type systems to both. This makes a simple notion of soundness possible: a type system is sound if, whenever an expression evaluates to a value, the value always has all of the types that the expression has.

There are reasonable notions of soundness that are stronger than this. We could follow [Mil78] and require that evaluation of a well-typed expression never "goes wrong", in the sense that semantic errors do not happen during evaluation. This would make the already tiresome proofs of soundness in this thesis even longer, so we shall instead stay with the weaker notion of soundness.

Evaluation will require substituting closed expressions for variables in expressions, so we need to define substitution before we define evaluation. Since we only allow substitution of a closed expression for a variable, we do not need to be concerned about variable capture.

Definition 2.1 *Substitution of an expression e for an object language variable x in an expression e' (written as $[e/x]e'$) is the expression consistent with the following rules:*

$$
\begin{array}{l}
[e/x]x = e \\
[e/x]y = y \textit{ if } y \neq x \\
[e/x]\texttt{fn } x\texttt{:}t \texttt{ => } e' = \texttt{fn } x\texttt{:}t \texttt{ => } e' \\
[e/x]\texttt{fn } y\texttt{:}t \texttt{ => } e' = \texttt{fn } y\texttt{:}t \texttt{ => } [e/x]e' \textit{ if } y \neq x \\
[e/x]\texttt{fix } x\texttt{:}t \texttt{ => fn } y\texttt{:}u \texttt{ => } e' = \\
\qquad \texttt{fix } x\texttt{:}t \texttt{ => fn } y\texttt{:}u \texttt{ => } e' \\
[e/x]\texttt{fix } f\texttt{:}t \texttt{ => fn } x\texttt{:}u \texttt{ => } e' = \\
\qquad \texttt{fix } f\texttt{:}t \texttt{ => fn } x\texttt{:}u \texttt{ => } e' \\
[e/x]\texttt{fix } f\texttt{:}t \texttt{ => fn } y\texttt{:}u \texttt{ => } e' = \\
\qquad \texttt{fix } f\texttt{:}t \texttt{ => fn } y\texttt{:}u \texttt{ => } [e/x]e' \textit{ if } y \neq x \textit{ and } f \neq x \\
[e/x]e_1\ e_2 = [e/x]e_1\ [e/x]e_2 \\
[e/x](e_1, \ \ldots, \ e_n) = ([e/x]e_1, \ \ldots, \ [e/x]e_n) \\
[e/x]() = () \\
[e/x]\texttt{elt\_}m\texttt{\_}n\ e' = \texttt{elt\_}m\texttt{\_}n\ [e/x]e' \\
[e/x]c\ e' = c\ [e/x]e' \\
[e/x]\texttt{case } e_0 \texttt{ of } c_1 \texttt{ => } e_1 \texttt{ | } \ldots \texttt{ | } c_n \texttt{ => } e_n \texttt{ end:}t = \\
\qquad \texttt{case } [e/x]e_0 \texttt{ of } c_1 \texttt{ => } [e/x]e_1 \texttt{ | } \ldots \texttt{ | } c_n \texttt{ => } [e/x]e_n \texttt{ end:}t
\end{array}
$$

For example, evaluating the expression

$$
\begin{array}{l}
(\texttt{fn double:}bool \to bool \texttt{ =>} \\
\quad (\texttt{double } (\texttt{fn x:}bool \texttt{ => x}) \ (\texttt{true } ()))) \\
(\texttt{fn f:}bool \to bool \texttt{ => fn x:}bool \texttt{ => f } (\texttt{f x}))
\end{array}
$$

would require computing the substitution

$$
\begin{array}{l}
[\texttt{fn f:}bool \to bool \texttt{ => fn x:}bool \texttt{ => f } (\texttt{f x})/\texttt{double}] \\
\quad (\texttt{double } (\texttt{fn x:}bool \texttt{ => x}) \ (\texttt{true } ()))
\end{array}
$$

which yields

$$
\begin{array}{l}
(\texttt{fn f:}bool \to bool \texttt{ => fn x:}bool \texttt{ => f } (\texttt{f x})) \\
\quad (\texttt{fn x:}bool \texttt{ => x}) \\
\quad (\texttt{true } ()).
\end{array}
$$

We will define evaluation only for closed expressions. This is convenient because it eliminates the need for an environment mapping variables to values.

ABS-SEM:
$$\frac{}{\texttt{fn}\ x{:}t\ \texttt{=>}\ e \Rightarrow \texttt{fn}\ x{:}t\ \texttt{=>}\ e}$$

APPL-SEM:
$$\frac{\begin{array}{c} e_1 \Rightarrow \texttt{fn}\ x{:}t\ \texttt{=>}\ e_3 \\ e_2 \Rightarrow v_2 \\ [v_1/x]e_3 \Rightarrow v_3 \end{array}}{e_1\ \ e_2 \Rightarrow v_3}$$

CONSTR-SEM:
$$\frac{e \Rightarrow v}{c\ \ e \Rightarrow c\ \ v}$$

CASE-SEM:
$$\frac{\begin{array}{c} \text{for some } i \text{ we have } e_0 \Rightarrow c_i\ \ v_i \\ e_i\ \ v_i \Rightarrow v \end{array}}{\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}t \Rightarrow v}$$

TUPLE-SEM:
$$\frac{\text{for } i \in 1 \ldots n \text{ we have } e_i \Rightarrow v_i}{(e_1,\ \ldots,\ e_n) \Rightarrow (v_1,\ \ldots,\ v_n)}$$

ELT-SEM:
$$\frac{e \Rightarrow (v_1,\ \ldots,\ v_n)}{\texttt{elt\_}m\texttt{\_}n\ \ e \Rightarrow v_m}$$

FIX-SEM:
$$\frac{}{\begin{array}{l}\texttt{fix}\ f{:}t\ \texttt{=>}\ \texttt{fn}\ x{:}u\ \texttt{=>}\ e \Rightarrow \\ \quad [\texttt{fix}\ f{:}t\ \texttt{=>}\ \texttt{fn}\ x{:}u\ \texttt{=>}\ e/f]\texttt{fn}\ x{:}u\ \texttt{=>}\ e\end{array}}$$

Figure 2.1: Monomorphic Semantics Rules

Our evaluation relation is written

$$e \Rightarrow v$$

which means that the closed expression e evaluates to the value v. The definition of the relation is in Figure 2.1. In some of the inference rules, we use the notation $n \ldots m$ to mean the set of integers between n and m.

For example, if we use T to abbreviate the derivation

$$\frac{\dfrac{}{() \Rightarrow ()}\,[\text{TUPLE-SEM}]}{\texttt{true}\ () \Rightarrow \texttt{true}\ ()}\,[\text{CONSTR-SEM}]$$

then the following is a valid evaluation, except to make the derivation fit on the page we omit the types after each variable binding and the -SEM suffix on the name of each semantics

rule:

$$\dfrac{\dfrac{\dfrac{}{(\texttt{fn x => x}) \Rightarrow (\texttt{fn x => x})}\,[\text{ABS}] \quad T \quad T}{(\texttt{fn x => x})\ (\texttt{true}\ ()) \Rightarrow (\texttt{true}\ ())}\,[\text{APPL}] \quad \dfrac{\dfrac{}{() \Rightarrow ()}\,[\text{TUPLE}]}{\texttt{nil}\ () \Rightarrow \texttt{nil}\ ()}\,[\text{CONSTR}]}{((\texttt{fn x => x})\ (\texttt{true}\ ()), \texttt{nil}\ ()) \Rightarrow (\texttt{true}\ (), \texttt{nil}\ ())}\,[\text{TUPLE}]}{\texttt{cons}\ ((\texttt{fn x => x})\ (\texttt{true}\ ()), \texttt{nil}\ ()) \Rightarrow \texttt{cons}\ (\texttt{true}\ (), \texttt{nil}\ ())}\,[\text{CONSTR}]$$

The CASE-SEM rule differs slightly from the closest analogy available in our syntax to true SML. In SML, `case` statements always evaluate the first case that applies. In this language, the order of the cases makes no difference; if multiple cases apply, then this semantics says the choice is made nondeterministically. For example, the expression

```
case true () of
   true => fn _ => true ()
 | true => fn _ => false ()
end : bool
```

evaluates to both `true ()` and `false ()`. This oddity could be avoided by requiring all of the constructors appearing in a `case` statement to be distinct, but we will have no need to require this.

This semantics does not formalize everything one might want to say about evaluation. A more stringent notion of soundness would allow evaluation of well typed expressions to fail to terminate, but it would not permit evaluation to get an error. Unfortunately expressions that do not terminate and expressions that get an error are not distinguished from each other by our semantics. Both kinds of expression have no value.

This could be repaired by adding a new value "wrong" along with rules that ensure that code with a type error evaluates to "wrong". As mentioned earlier, we will not take this route because of the added tedium.

2.5 ML Typing

The system described in this section checks that the ML types embedded in an expression are consistent with each other, and it determines an ML type for the expression as a whole. The ML type of an expression depends on the assumptions we make about the ML types of constructors used in the expression, so we shall discuss that first.

If c is a value constructor, we say that c maps values of type t to elements of the datatype tc by writing

$$c \overset{\text{def}}{::} t \hookrightarrow tc\,.$$

For example, the effect of the SML datatype declaration

$$\texttt{datatype}\ \mathit{blist}\ \texttt{=}\ \texttt{nil of}\ \mathit{tunit}\ \texttt{|}\ \texttt{cons of}\ \mathit{bool} * \mathit{blist}$$

on the SML environment would be analogous to adding these assumptions to the environment:

$$\texttt{nil} \overset{\text{def}}{::} \mathit{tunit} \hookrightarrow \mathit{blist}$$
$$\texttt{cons} \overset{\text{def}}{::} \mathit{bool} * \mathit{blist} \hookrightarrow \mathit{blist}$$

The examples in this chapter will also make these assumptions:

$$\texttt{true} \overset{\text{def}}{::} \mathit{tunit} \hookrightarrow \mathit{bool}$$
$$\texttt{false} \overset{\text{def}}{::} \mathit{tunit} \hookrightarrow \mathit{bool}$$

In general, for a type system that uses assumptions about value constructors to make sense, we need the assumptions to be consistent in certain ways. For the type system described in this section, all we need to know is that we have exactly one assumption about each constructor:

Assumption 2.2 (Constructors have Unique ML Types) *For each c, there are unique t and tc such that*

$$c \overset{\text{def}}{::} t \hookrightarrow \mathit{tc}$$

The ML type of an expression depends on the ML types we assign to the free variables appearing in the expression, so our typing relation will describe the type of an expression given a partial function VMVM from variables to ML types.

The name VM is an example of a naming convention that will be used for all of the partial functions used as environments in this thesis. Each name has two letters. The first letter stands for the domain (V stands for "variable") and the second letter stands for the codomain (M stands for "ML type"). In later chapters, M will sometimes stand for "ML type scheme", but since the formal language is monomorphic, it just stands for "ML type" here.

We use the notation $\mathrm{VM}[x := t]$ to mean the partial map identical to VM everywhere except at x, which it maps to t. The notation $\cdot$ means the partial map that is undefined everywhere.

The ML typing relation is written as

$$\mathrm{VM} \vdash e :: t$$

which means that assuming that all free variables x in e have the type $\mathrm{VM}(x)$, then e has the ML type t. The definition of this relation is in Figure 2.2. These rules are similar to the rules in Mini-ML [CDDK86], except we have no polymorphism and we separate the operator for destructuring tuples from the operator for forming abstractions.

If this type system is sound, then we would expect that a closed expression has a type and it evaluates to a value then the value has the same type. We can formally state this as follows:

VAR-VALID:
$$\frac{\mathrm{VM}(x) = t}{\mathrm{VM} \vdash x :: t}$$

ABS-VALID:
$$\frac{\mathrm{VM}[x := t_1] \vdash e :: t_2}{\mathrm{VM} \vdash (\texttt{fn}\ x : t_1\ \texttt{=>}\ e) :: t_1 \to t_2}$$

APPL-VALID:
$$\frac{\mathrm{VM} \vdash e_1 :: t_2 \to t_1 \qquad \mathrm{VM} \vdash e_2 :: t_2}{\mathrm{VM} \vdash e_1\ e_2 :: t_1}$$

CONSTR-VALID:
$$\frac{c \overset{\text{def}}{::} t \hookrightarrow tc \qquad \mathrm{VM} \vdash e :: t}{\mathrm{VM} \vdash c\ e :: tc}$$

CASE-VALID:
$$\frac{\begin{array}{c}\mathrm{VM} \vdash e_0 :: tc \\ \text{for all } i \text{ we have } c_i \overset{\text{def}}{::} t_i \hookrightarrow tc \\ \text{for all } i \text{ we have } \mathrm{VM} \vdash e_i :: t_i \to u\end{array}}{\mathrm{VM} \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}\ u) :: u}$$

TUPLE-VALID:
$$\frac{\text{for all } i \text{ we have } \mathrm{VM} \vdash e_i :: t_i}{\mathrm{VM} \vdash (e_1, \ldots, e_m) :: t_1 * \ldots * t_m}$$

ELT-VALID:
$$\frac{\mathrm{VM} \vdash e :: t_1 * \ldots * t_n}{\mathrm{VM} \vdash \texttt{elt\_}m\texttt{\_}n\ e :: t_m}$$

FIX-VALID:
$$\frac{\mathrm{VM}[f := t_1 \to t_2] \vdash (\texttt{fn}\ x : t_1\ \texttt{=>}\ e) :: t_1 \to t_2}{\mathrm{VM} \vdash (\texttt{fix}\ f : t_1 \to t_2\ \texttt{=>}\ \texttt{fn}\ x : t_1\ \texttt{=>}\ e) :: t_1 \to t_2}$$

Figure 2.2: Monomorphic ML Typing Rules

Fact 2.3 (ML Type Soundness) *If* $\cdot \vdash e :: t$ *and* $e \Rightarrow v$ *then* $\cdot \vdash v :: t$.

We will not prove this. A partially mechanically verified proof of this theorem for a similar language is in [MP91].

We intend our ML type system to be an unambiguous framework that supports the refinement type system. The following theorem states that it is unambiguous. Theorem 2.54 (Inferred Types Refine) on page 68 states in what sense the refinement type system is supported by the ML type system.

Lemma 2.4 (Unique Inferred ML Types) *If* $\mathrm{VM} \vdash e :: t$ *and* $\mathrm{VM} \vdash e :: t'$ *then* $t = t'$.

Proof: By straightforward induction on e.

Case: $e = x$ Then the last inference in both of our hypotheses is VAR-VALID, with the premises $\text{VM}(x) = t$ and $\text{VM}(x) = t'$. Thus $t = t'$.

Case: $e = \texttt{fn}\ x : t_1\ \texttt{=>}\ e'$ Then the last inference in both of our hypotheses is ABS-VALID. Since the ML type t_1 is explicit in the syntax, we must have $t = t_1 \to t_2$ and $t' = t_1 \to t_2'$. The premises of the two uses of ABS-VALID must be

$$\text{VM}[x := t_1] \vdash e' :: t_2$$

and

$$\text{VM}[x := t_1] \vdash e' :: t_2'.$$

The induction hypothesis gives $t_2 = t_2'$, so we must have $t = t'$.

Case: $e = e_1\ e_2$ Then the last inference in our hypotheses must be APPL-VALID with the premises

$$\text{VM} \vdash e_1 :: t_2 \to t$$

and

$$\text{VM} \vdash e_1 :: t_2' \to t',$$

among others. The induction hypothesis applied to these gives $t_2 \to t = t_2' \to t'$, which implies $t = t'$.

Case: $e = c\ e$ Then the last inference in our hypotheses must be CONSTR-VALID with the premises

$$c \overset{\text{def}}{::} u \hookrightarrow tc$$

and

$$c \overset{\text{def}}{::} u' \hookrightarrow tc',$$

among others, where $t = tc$ and $t' = tc'$. Assumption 2.2 (Constructors have Unique ML Types) on page 26 gives $tc = tc'$, which implies $t = t'$.

Case: $e = \texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}\ u$
The last inference of both of our hypotheses must be CASE-VALID, which immediately gives $t = u$ and $t' = u$, so $t = t'$.

Case: $e = (e_1, \ldots, e_n)$ Then the last inference in the derivation of each of our hypotheses must be TUPLE-VALID with the premises

$$\text{for all } i \text{ we have } \text{VM} \vdash e_i :: t_i$$

and

$$\text{for all } i \text{ we have } \text{VM} \vdash e_i :: t_i'$$

where $t = t_1 * \ldots * t_n$ and $t' = t'_1 * \ldots * t'_n$. Our induction hypothesis gives $t_i = t'_i$ for all i, so $t = t'$.

Case: $e =$ `elt_`m`_`n `e'` Then the last inference in the derivation of each of our conclusions must be ELT-VALID with the premises

$$\text{VM} \vdash e' :: t_1 * \ldots * t_n$$

and

$$\text{VM} \vdash e' :: t'_1 * \ldots * t'_n$$

where $t = t_m$ and $t' = t'_m$. Our induction hypothesis gives $t_1 * \ldots * t_n = t'_1 * \ldots * t'_n$, so we must have $t = t'$.

Case: $e =$ `fix` $f : t_1 \to t_2$ `=> fn` $x : t_1$ `=>` e' Then the last inference in the derivation of each of our conclusions must be FIX-VALID. This immediately gives $t = t_1 \to t_2$ and $t' = t_1 \to t_2$, which implies $t = t'$. □

If an expression has an ML type, then all of its free variables must be bound to an ML type in the environment. This will be important later on when we are proving things about the refinement type system because the refinement type rule for `case` statements has a premise requiring the `case` statement to have an ML type. To put it formally,

Fact 2.5 (ML Free Variables Bound) *If* $\text{VM} \vdash e :: t$ *and* x *is free in* e*, then* $\text{VM}(x)$ *is defined.*

Proof of this would be by induction on the derivation of $\text{VM} \vdash e :: t$.

One step along the path to proving Fact 2.3 (ML Type Soundness) on page 27 is showing that there is a natural kind of substitution on ML type derivations that preserves soundness. We will use this once in Lemma 2.70 (Value Substitution) on page 93, which is the refinement type analogue of this fact. The use is in the CASE-TYPE case of that lemma.

Fact 2.6 (ML Value Substitution) *If* $\text{VM} \vdash e_1 :: t_1$ *and* $\text{VM}[x := t_1] \vdash e_2 :: t_2$ *then* $\text{VM} \vdash [e_1/x]e_2 : t_2$.

Proof of this would be a straightforward induction on the derivation of $\text{VM}[x := t_1] \vdash e_2 :: t_2$.

2.6 Monomorphic Refinement Types

Now that we have given our version of the ML type system, we can give an analogous description of refinement types for the same expressions. We shall use r, k, and p as metavariables standing for refinement types.

Syntactically, refinement types have an intersection operator "$\wedge$". This is the only difference in structure between the syntax for refinement types and the syntax for ML types, so we can define the syntax for refinement types with the grammar

$$r ::= r \wedge r \mid r \to r \mid rc \mid r * \ldots * r \mid runit.$$

Once again we have a special name for the empty tuple type; this time it is $runit$. Because we give different names to $tunit$ and $runit$, inspecting a type tells us immediately whether it is an ML type or a refinement type.

In our concrete syntax we shall adopt the convention that $\to$ binds tighter than $\wedge$. This makes it easy to write types consisting of an intersection of many arrow types. Since the principal type of each function has this form, being able to write these concisely is convenient. For instance, the type of boolean negation is $(tt \to \mathit{ff}) \wedge (\mathit{ff} \to tt)$, which we can write as $tt \to \mathit{ff} \wedge \mathit{ff} \to tt$.

The set of refinement types an expression may have depends on its ML type. For instance, an expression with ML type $bool \to bool$ may have refinement type $tt \to tt$ or $\top_{bool} \to \mathit{ff}$, but not tt. We write this as

$$tt \to tt \sqsubset bool \to bool$$
$$\top_{bool} \to \mathit{ff} \sqsubset bool \to bool$$

but not

$$tt \sqsubset bool \to bool.$$

The assertion $r \sqsubset t$ can be read aloud as "r refines t"; hence the name "refinement types". We will call a refinement type that refines no ML type "malformed".

Before we can formally define the $\sqsubset$ relation between refinement types and ML types, we have to make some assumptions about which refinement type constructors refine which ML type constructors. We shall write the assumption that a refinement type constructor rc refines an ML type constructor tc as

$$rc \overset{\text{def}}{\sqsubset} tc.$$

For example, after we formally define $\sqsubset$, our derivation of $\top_{bool} \to \mathit{ff} \sqsubset bool \to bool$ will use the assumptions

$$\top_{bool} \overset{\text{def}}{\sqsubset} bool$$

and

$$\mathit{ff} \overset{\text{def}}{\sqsubset} bool.$$

The examples below will also use these assumptions:

$$tt \overset{\text{def}}{\sqsubset} bool$$
$$\bot_{bool} \overset{\text{def}}{\sqsubset} bool$$

For our soundness proof to go through, we will need the $\overset{\text{def}}{\sqsubset}$ relation to be well-behaved in the following sense:

$$\text{AND-REF:} \quad \frac{r_1 \sqsubset t \qquad r_2 \sqsubset t}{r_1 \wedge r_2 \sqsubset t}$$

$$\text{ARROW-REF:} \quad \frac{r_1 \sqsubset t_1 \qquad r_2 \sqsubset t_2}{r_1 \to r_2 \sqsubset t_1 \to t_2}$$

$$\text{RCON-REF:} \quad \frac{rc \overset{\text{def}}{\sqsubset} tc}{rc \sqsubset tc}$$

$$\text{TUPLE-REF:} \frac{\text{for } i \text{ in } 1 \ldots n \text{ we have } r_i \sqsubset t_i}{r_1 * \ldots * r_n \sqsubset t_1 * \ldots * t_n}$$

Figure 2.3: Monomorphic Refinement Rules

Assumption 2.7 (Unique Predefined Refinements) *For all rc there is a unique tc such that* $rc \overset{\text{def}}{\sqsubset} tc$.

Also, for there to be any hope of manipulating refinement types with an algorithm, the set of refinements of any ML type constructor must be finite:

Assumption 2.8 (Finite Predefined Refinements) *For all tc, the set* $\{rc \mid rc \overset{\text{def}}{\sqsubset} tc\}$ *is finite.*

We formally define the $\sqsubset$ relation in Figure 2.3.

TUPLE-REF implies $runit \sqsubset tunit$ because we can choose $n = 0$, and $runit$ and $tunit$ are our names for the empty tuples of refinement types and ML types, respectively.

Refinement types that refine some ML type are generally easier to reason about than refinement types that do not:

Definition 2.9 (Well-formed Refinement Type) *We say that a refinement type* r *is* well-formed *if there is an ML type* t *such that* $r \sqsubset t$. *Otherwise we say it is* ill-formed.

From the rule defining $\sqsubset$, it follows that each refinement type refines at most one ML type. Stating this formally,

Lemma 2.10 (Unique ML types) *If* $r \sqsubset t$ *and* $r \sqsubset u$ *then* $t = u$.

The proof of this is straightforward.

Proof: By induction on r.

Case: $r = k \wedge p$ The only way to derive $r \sqsubset t$ is to use AND-REF where one of the premises is $k \sqsubset t$. Similarly, the only way to derive $r \sqsubset u$ is to use AND-REF where one of the premises is $k \sqsubset u$. Applying the induction hypothesis to these two premises gives $t = u$, which is our conclusion.

Case: $r = rc$ Assumption 2.7 (Unique Predefined Refinements) on page 31 gives our conclusion directly.

Case: $r = r_1 * \ldots * r_n$ We can only derive $r \sqsubset t$ by using TUPLE-REF. Thus t must have the form $t_1 * \ldots * t_n$, and from the premises of TUPLE-REF we know

$$\text{for } i \text{ between 1 and } n \text{ we have } r_i \sqsubset t_i.$$

Similarly, $r \sqsubset u$ tells us that u has the from $u_1 * \ldots * u_n$ and

$$\text{for } i \text{ between 1 and } n \text{ we have } r_i \sqsubset u_i.$$

Using the induction hypothesis gives

$$\text{for } i \text{ between 1 and } n \text{ we have } t_i = u_i.$$

Thus $t = u$.

Case: $r = runit$ Then the only way to derive our hypotheses is by using UNIT-REF, and $t = u = tunit$.

Case: $r = r_1 \to r_2$ Then the last premise of the derivation of $r \sqsubset t$ must be ARROW-REF, so t must have the form $t_1 \to t_2$ and the premises of ARROW-REF must be $r_1 \sqsubset t_1$ and $r_2 \sqsubset t_2$. Similarly, $r \sqsubset u$ tells us that u has the form $u_1 \to u_2$ and $r_1 \sqsubset u_1$ and $r_2 \sqsubset u_2$. The induction hypothesis tells us that $t_1 = u_1$ and $t_2 = u_2$, so $t = u$. □

Since each refinement type refines at most one ML type, we can define a partial function that maps each refinement type to the corresponding ML type, if there is one.

Definition 2.11 *If $r \sqsubset t$ then we say $t = \text{rtom}(r)$. If there is no t such that $r \sqsubset t$, then* $\text{rtom}(r)$ *is undefined.*

The name rtom stands for "Refinement to ML". We extend this in the natural way to work on environments: $\text{rtom}(\text{VM})(x) = \text{rtom}(\text{VM}(x))$.

As one would expect, if we know which ML type is refined by a refinement type, that heavily constrains the form of the refinement type. For example, we have

Fact 2.12 (Tuple Refines) *If $r \sqsubset t_1 * \ldots * t_h$ then r has the form*

$$r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn}.$$

Proof of this would be a trivial induction on the derivation of $r \sqsubset t_1 * \ldots * t_h$. We will use this in Lemma 2.26 (Tuple Subtyping) on page 42.

2.6.1 Subtyping

If two refinement types refine the same ML type, then it makes sense to compare them. Our comparison operator is written $\leq$. For instance, in the presence of reasonable assumptions about our refinement type constructors, the following assertions are true:

$$\begin{gathered} tt \leq \top_{bool} \\ tt \wedge \mathit{ff} \leq \bot_{bool} \\ \bot_{bool} \leq tt \wedge \mathit{ff} \\ (tt * \mathit{ff}) \wedge (\mathit{ff} * tt) \leq (\bot_{bool} * \bot_{bool}) \\ tt \to \mathit{ff} \wedge \mathit{ff} \to tt \leq tt \to \top_{bool} \end{gathered}$$

and these assertions are false:

$$\begin{gathered} \mathit{ff} \leq tt \\ tt \to \mathit{ff} \leq \mathit{ff} \to tt. \end{gathered}$$

The rules defining $\leq$ must take into account some assumptions about how the refinement type constructors behave. We need to know that some refinement type constructors are subtypes of others, which we shall write as

$$rc_1 \overset{\text{def}}{\leq} rc_2.$$

We also need to be able to compute intersections of refinement type constructors, if they both refine the same ML type constructor. We write this as a partial binary operation $\overset{\text{def}}{\wedge}$ on refinement type constructors. For example, the definition of $\leq$ we will give below allows us to derive

$$tt \to tt \wedge \top_{bool} \to \mathit{ff} \leq \bot_{bool} \to \bot_{bool},$$

and the derivation uses these assumptions:

$$\begin{gathered} \bot_{bool} \overset{\text{def}}{\leq} tt \\ \bot_{bool} \overset{\text{def}}{\leq} \top_{bool} \\ tt \overset{\text{def}}{\wedge} \mathit{ff} \overset{\text{def}}{\leq} \bot_{bool}. \end{gathered}$$

For our definition of subtyping to make sense, we need $\overset{\text{def}}{\leq}$ and $\overset{\text{def}}{\wedge}$ to be consistent in certain ways. First we need transitivity and reflexivity of $\overset{\text{def}}{\leq}$:

Assumption 2.13 (reflex-$\overset{\text{def}}{\leq}$) *For all rc we have* $rc \overset{\text{def}}{\leq} rc$.

Assumption 2.14 (trans-$\overset{\text{def}}{\leq}$) *If $rc_1 \overset{\text{def}}{\leq} rc_2$ and $rc_2 \overset{\text{def}}{\leq} rc_3$ then $rc_1 \overset{\text{def}}{\leq} rc_3$.*

If two refinement type constructors are comparable, they must refine the same ML type constructor:

Assumption 2.15 (Refines $\overset{\text{def}}{\leq}$) *If $rc \overset{\text{def}}{\leq} kc$ then $rc \overset{\text{def}}{\sqsubset} tc$ if and only if $kc \overset{\text{def}}{\sqsubset} tc$.*

We need to know $\overset{\text{def}}{\wedge}$ is defined for refinements of the same ML type constructor, and it is a greatest lower bound in the set of those refinements:

Assumption 2.16 ($\overset{\text{def}}{\wedge}$-defined) *If $rc \overset{\text{def}}{\sqsubset} tc$ and $kc \overset{\text{def}}{\sqsubset} tc$ then $rc \overset{\text{def}}{\wedge} kc$ is defined.*

Assumption 2.17 ($\overset{\text{def}}{\wedge}$ Elim) *If $rc \overset{\text{def}}{\wedge} kc$ is defined, then $rc \overset{\text{def}}{\wedge} kc \overset{\text{def}}{\leq} rc$ and $rc \overset{\text{def}}{\wedge} kc \overset{\text{def}}{\leq} kc$.*

Assumption 2.18 (and-intro-$\overset{\text{def}}{\leq}$) *If $rc \overset{\text{def}}{\leq} kc$ and $rc \overset{\text{def}}{\leq} pc$ then $rc \overset{\text{def}}{\leq} (kc \overset{\text{def}}{\wedge} pc)$.*

Our subtyping operator $\leq$ is defined by the rules in Figure 2.4. Several of these rules need to be explained:

Since $runit$ is our name for the empty tuple, we interpret the rule for dealing with tuples so they apply to $runit$ also.

Some of the rules resemble each other. The rules ARROW-SUB, TUPLE-SUB, and RCON-SUB are similar, as are ARROW-AND-ELIM-SUB, TUPLE-AND-ELIM-SUB, and RCON-AND-ELIM-SUB. In Chapter 5 we will change the syntax for refinement types so that arrows, tuples, and monomorphic refinement type constructors are all a special case of polymorphic refinement type constructors. After we do that, each triplet of similar rules will collapse to one rule.

The rule ARROW-SUB is conventional for systems with subtypes, although it is often surprising to the uninitiated. As the type on the right side of the arrow gets larger, the entire type gets larger. However, as the type on the left side gets larger, the entire type gets smaller. Another way to say this is that arrow is contravariant in its first argument and covariant in its second argument.

To understand this it helps to think of refinement types as sets and to read "$\leq$" as subset. An arrow type $r_1 \rightarrow r_2$ means the set of all functions that map all elements of the set r_1 to elements of r_2. If int is the set of all integers and ev is the set of all even integers, then the following subtype relations are true in our model:

$$ev \leq int$$
$$ev \rightarrow ev \leq ev \rightarrow int$$
$$int \rightarrow ev \leq ev \rightarrow ev$$

SELF-SUB: $$\frac{r \sqsubset t}{r \leq r}$$

AND-ELIM-R-SUB: $$\frac{r \sqsubset t \qquad k \sqsubset t}{r \wedge k \leq r}$$

AND-ELIM-L-SUB: $$\frac{r \sqsubset t \qquad k \sqsubset t}{r \wedge k \leq k}$$

AND-INTRO-SUB: $$\frac{r \leq k_1 \qquad r \leq k_2}{r \leq k_1 \wedge k_2}$$

TRANS-SUB: $$\frac{r \leq p \qquad p \leq k}{r \leq k}$$

ARROW-SUB: $$\frac{k_1 \leq r_1 \qquad r_2 \leq k_2}{r_1 \to r_2 \leq k_1 \to k_2}$$

ARROW-AND-ELIM-SUB: $$\frac{r_1 \to (r_2 \wedge r_3) \sqsubset t}{r_1 \to r_2 \wedge r_1 \to r_3 \leq r_1 \to (r_2 \wedge r_3)}$$

RCON-SUB: $$\frac{rc \overset{\text{def}}{\leq} kc}{rc \leq kc}$$

RCON-AND-ELIM-SUB: $$\frac{rc_1 \overset{\text{def}}{\wedge} rc_2 \sqsubset t}{rc_1 \wedge rc_2 \leq rc_1 \overset{\text{def}}{\wedge} rc_2}$$

TUPLE-SUB: $$\frac{\text{for all } i \text{ we have } r_i \leq k_i}{r_1 * \ldots * r_n \leq k_1 * \ldots * k_n}$$

TUPLE-AND-ELIM-SUB: $$\frac{(r_1 \wedge r_1') * \ldots * (r_n \wedge r_n') \sqsubset t}{(r_1 * \ldots * r_n) \wedge (r_1' * \ldots * r_n') \leq (r_1 \wedge r_1') * \ldots * (r_n \wedge r_n')}$$

Figure 2.4: Monomorphic Subtyping Rules

Thus the intuitive model is consistent with the inference rule.

Following [Pie91b], we use subtyping inference rules to express the fact that intersection is a greatest lower bound. The rules AND-ELIM-L-SUB and AND-ELIM-R-SUB ensure that intersection is a lower bound and AND-INTRO-SUB guarantees that it is a greatest lower bound. Since intersection is a greatest lower bound, it is commutative, associative, and monotone in both arguments. The usual proofs that any greatest lower bound has these properties translate directly into uses of the inference rules. For example, here is a proof that intersection is monotone in its first argument:

Lemma 2.19 *If* $r_1 \leq r_2$ *and* $r_1 \wedge r_2 \wedge r_3 \sqsubset t$*, then* $r_1 \wedge r_3 \leq r_2 \wedge r_3$.

Proof: The only way to derive $r_1 \wedge r_2 \wedge r_3 \sqsubset t$ is by repeatedly using AND-REF, so we must have $r_1 \sqsubset t$ and $r_2 \sqsubset t$ and $r_3 \sqsubset t$. The rule AND-ELIM-R-SUB gives $r_1 \wedge r_3 \leq r_1$. Applying TRANS-SUB to this and our hypothesis gives $r_1 \wedge r_3 \leq r_2$. The rule AND-ELIM-L-SUB gives $r_1 \wedge r_3 \leq r_3$. The previous two assertions and AND-INTRO-SUB give $r_1 \wedge r_3 \leq r_2 \wedge r_3$, which is what we wanted to show. □

Once we have a subtying relation, we can define a natural notion of equivalence:

Definition 2.20 *We say that* r_1 *is equivalent to* r_2*, or in symbols* $r_1 \equiv r_2$*, if* $r_1 \leq r_2$ *and* $r_2 \leq r_1$.

This relation is an equivalence relation on the refinements of any ML type, but it is only a partial equivalence relation on refinement types as a whole because some refinement types refine no ML type. For example, the refinement type $tt \wedge tt \to tt$ is not equivalent to itself according to this definition.

The subtyping rules in Figure 2.4 ensure that the types involved are well behaved in the following sense:

Theorem 2.21 (Subtypes Refine) *If* $r \leq k$*, then there is a unique ML type* t *such that* $r \sqsubset t$ *and* $k \sqsubset t$.

Proof: By Lemma 2.10 (Unique ML Types) on page 31, there is at most one t such that $r \sqsubset t$ and $k \sqsubset t$, so all we need to show here is that there is at least one such t. We do this by induction on the derivation of $r \leq k$.

Case: SELF-SUB Then $r = k$ and the premise of SELF-SUB gives a t such that $r \sqsubset t$.

Case: AND-ELIM-R-SUB Then r has the form $k \wedge p$ and the premises of AND-ELIM-R-SUB must be

$$k \sqsubset t \tag{2.1}$$

and

$$p \sqsubset t. \tag{2.2}$$

Applying AND-REF to these gives

$$k \wedge p \sqsubset t. \tag{2.3}$$

Our conclusions are (2.1) and (2.3).

Case: AND-ELIM-L-SUB Similar to AND-ELIM-R-SUB.

Case: AND-INTRO-SUB Then k has the form $k_1 \wedge k_2$ and the premises of AND-INTRO-SUB must be

$$r \leq k_1 \tag{2.4}$$

and

$$r \leq k_2 \tag{2.5}$$

Using the induction hypothesis on (2.4) gives a t such that

$$r \sqsubset t \tag{2.6}$$

and

$$k_1 \sqsubset t. \tag{2.7}$$

Using the induction hypothesis on (2.5) gives a u such that

$$r \sqsubset u \tag{2.8}$$

and

$$k_2 \sqsubset u \tag{2.9}$$

Lemma 2.10 (Unique ML Types) on page 31 applied to (2.6) and (2.8) gives $t = u$, so we can use AND-REF to combine (2.7) and (2.9) to get

$$k_1 \wedge k_2 \sqsubset t.$$

This and (2.6) are our conclusions.

Case: TRANS-SUB Then the premises of TRANS-SUB are $r \leq p$ and $p \leq k$. Applying the induction hypothesis to both of these gives t and u such that all of the following hold:

$$\begin{array}{l} r \sqsubset t \\ p \sqsubset t \\ p \sqsubset u \\ k \sqsubset u. \end{array}$$

Unique ML Types applied to the middle two gives us $t = u$, so the first and the last are our conclusions.

Case: RCON-SUB Then $r = rc$ and $k = kc$ and the premise of RCON-SUB is $rc \overset{\text{def}}{\leq} kc$. Assumption 2.7 (Unique Predefined Refinements) on page 31 gives a tc such that $rc \overset{\text{def}}{\sqsubset} tc$. By Assumption 2.15 (Refines $\overset{\text{def}}{\leq}$) on page 34 and $rc \overset{\text{def}}{\leq} kc$ we have $kc \overset{\text{def}}{\sqsubset} tc$. RCON-REF gives $rc \sqsubset tc$ and $kc \sqsubset tc$, which are our conclusions.

Case: RCON-AND-ELIM-SUB Then $r = rc_1 \wedge rc_2$ and $k = rc_1 \overset{\text{def}}{\wedge} rc_2$. The premise of RCON-AND-ELIM-SUB is

$$rc_1 \overset{\text{def}}{\wedge} rc_2 \sqsubset t. \tag{2.10}$$

By Assumption 2.17 ($\overset{\text{def}}{\wedge}$ Elim) on page 34 and Assumption 2.15 (Refines $\overset{\text{def}}{\leq}$) on page 34, $rc_1 \overset{\text{def}}{\sqsubset} t$ and $rc_2 \overset{\text{def}}{\sqsubset} t$. Because "$\overset{\text{def}}{\sqsubset}$" only relates refinement type constructors to ML type constructors, t must have the form tc. By Assumption 2.17 ($\overset{\text{def}}{\wedge}$ Elim) on page 34, we know $rc_1 \overset{\text{def}}{\wedge} rc_2 \overset{\text{def}}{\leq} rc_1$. By Assumption 2.15 (Refines $\overset{\text{def}}{\leq}$) on page 34, it follows that $rc_1 \overset{\text{def}}{\wedge} rc_2 \overset{\text{def}}{\sqsubset} tc$. By RCON-SUB,

$$rc_1 \overset{\text{def}}{\wedge} rc_2 \sqsubset tc.$$

This and (2.10) are our conclusions.

Case: ARROW-SUB Then $r = r_1 \to r_2$ and $k = k_1 \to k_2$ and the premises of ARROW-SUB are

$$k_1 \leq r_1$$

and

$$r_2 \leq k_2.$$

Applying the induction hypothesis to both of these gives t_1 and t_2 such that:

$$\begin{aligned} k_1 &\sqsubset t_1 \\ r_1 &\sqsubset t_1 \\ r_2 &\sqsubset t_2 \\ k_2 &\sqsubset t_2. \end{aligned}$$

Applying ARROW-REF to these gives

$$r_1 \to r_2 \sqsubset t_1 \to t_2$$

and

$$k_1 \to k_2 \sqsubset t_1 \to t_2,$$

which are our conclusions.

Case: ARROW-AND-ELIM-SUB Then $r = r_1 \to (r_2 \wedge r_3)$ and $k = r_1 \to r_2 \wedge r_1 \to r_3$. The premise of ARROW-AND-ELIM-SUB is

$$r_1 \to (r_2 \wedge r_3) \sqsubset t. \tag{2.11}$$

The last inferences of the derivation of this must be ARROW-REF and AND-REF, so we must have

$$t = t_1 \to t_2$$
$$r_1 \sqsubset t_1$$
$$r_2 \sqsubset t_2$$
$$r_3 \sqsubset t_2$$

Applying ARROW-REF and AND-REF to these in a different order gives

$$r_1 \to r_2 \wedge r_1 \to r_3 \sqsubset t_1 \to t_2$$

This and (2.11) are our conclusions.

Case: TUPLE-SUB Then $r = r_1 * \ldots * r_n$ and $k = k_1 * \ldots * k_n$ and the premise of TUPLE-SUB is $r_i \leq k_i$ for all i between 1 and n. Applying the induction hypothesis to this gives, for each i between 1 and n, a t_i such that $r_i \sqsubset t_i$ and $k_i \sqsubset t_i$. TUPLE-REF gives

$$r_1 * \ldots * r_n \sqsubset t_1 * \ldots * t_n$$

and

$$k_1 * \ldots * k_n \sqsubset t_1 * \ldots * t_n,$$

which are our conclusions. If we take $n = 0$, this conclusion tells us $runit \sqsubset tunit$, which is true and unremarkable.

Case: TUPLE-AND-ELIM-SUB Then $r = (r_1 * \ldots * r_n) \wedge (r'_1 * \ldots * r'_n)$ and $k = (r_1 \wedge r'_1) * \ldots * (r_n \wedge r'_n)$. The premise of TUPLE-AND-ELIM-SUB is

$$(r_1 \wedge r'_1) * \ldots * (r_n \wedge r'_n) \sqsubset t. \tag{2.12}$$

The only way to derive this is with TUPLE-REF, so we must have $t = t_1 * \ldots * t_n$ and

$$(r_i \wedge r'_i) \sqsubset t_i$$

for i between 1 and n. Each of these assumptions must follow from AND-REF, so for all i we must have

$$r_i \sqsubset t_i$$

and

$$r'_i \sqsubset t_i.$$

Applying TUPLE-REF to these gives

$$r_1 * \ldots * r_n \sqsubset t$$

and

$$r'_1 * \ldots * r'_n \sqsubset t.$$

Applying AND-REF to these gives

$$(r_1 * \ldots * r_n) \wedge (r'_1 * \ldots * r'_n) \sqsubset t.$$

This and (2.12) are our conclusions. If we take $n = 0$, our conclusions are $runit \wedge runit \sqsubset t$ and $runit \sqsubset t$, both of which are true and uninteresting. □

Some uses of $\wedge$ are inessential. We do not need to be able to take intersections of tuples or of refinement type constructors; every intersection of tuples can be simplified to a tuple of intersections, and every intersection of refinement type constructors can be simplified to a refinement type constructor. These simplifications are necessary in many places in the proofs appearing later in this chapter, so we will prove that they are valid now.

For example,

$$(tt * \top_{bool}) \wedge (ff * ff) \equiv \bot_{bool} * ff$$

and

$$tt \wedge \top_{bool} \equiv tt.$$

We will prove that simplifications like this are possible in the general case.

Lemma 2.22 (Tuple Intersection) *If* $r_1 * \ldots * r_n \sqsubset t$ *and* $k_1 * \ldots * k_n \sqsubset t$ *then*

$$(r_1 * \ldots * r_n) \wedge (k_1 * \ldots * k_n) \equiv (r_1 \wedge k_1) * \ldots * (r_n \wedge k_n).$$

Proof of $(r_1 * \ldots * r_n) \wedge (k_1 * \ldots * k_n) \leq (r_1 \wedge k_1) * \ldots * (r_n \wedge k_n)$**:** Immediate from TUPLE-AND-ELIM-SUB.

Proof of $(r_1 \wedge k_1) * \ldots * (r_n \wedge k_n) \leq (r_1 * \ldots * r_n) \wedge (k_1 * \ldots * k_n)$**:** Use AND-ELIM-L-SUB and AND-ELIM-R-SUB to get

$$\text{for } h \text{ in } 1 \ldots n \text{ we have } r_h \wedge k_h \leq r_h$$

and

$$\text{for } h \text{ in } 1 \ldots n \text{ we have } r_h \wedge k_h \leq k_h.$$

Then TUPLE-SUB gives

$$(r_1 \wedge k_1) * \ldots * (r_n \wedge k_n) \leq r_1 * \ldots * r_n$$

and

$$(r_1 \wedge k_1) * \ldots * (r_n \wedge k_n) \leq k_1 * \ldots * k_n.$$

Finally AND-INTRO-SUB gives

$$(r_1 \wedge k_1) * \ldots * (r_n \wedge k_n) \leq (r_1 * \ldots * r_n) \wedge (k_1 * \ldots * k_n),$$

which is our conclusion. □

In a moment, we will present a simple algorithm that simplifies tuple refinement types. Since this is the first algorithm we present at the meta-level (that is, it manipulates refinement types as objects), we need to describe the notation we will use for these algorithms first.

The notation is basically Standard ML, except we allow free use of set notation and ellipses "...", provided the meaning is unambiguous. Since sets in mathematics and records in SML are both written with braces, we must give up one or the other to avoid ambiguity; we give up records. Our meta-level algorithms will also occasionally lapse into English or mathematics. As an example, we can give the simple algorithm for simplifying tuples:

$$\texttt{fun tuplesimp } (r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn}) \texttt{ =} \\ (r_{11} \wedge \ldots \wedge r_{1n}) * \ldots * (r_{h1} \wedge \ldots \wedge r_{hn})$$

This algorithm uses SML's destructuring convention with ellipses to simultaneously bind h and n to nonnegative integers and the variables r_{ij} to refinement types for i between 1 and h and j between 1 and n. Then it uses ellipses again to construct a refinement type that is a rearranged form of the given refinement type.

This notation has advantages and disadvantages. Since it is based on a real programming language, it tends to remain comprehensible as the algorithms we describe get more complex. Since it is based on Standard ML, it is likely to be understandable to people reading this thesis. However, basing the metalanguage on SML also invites confusion between the metalanguage and the object language, and this form of metalanguage is not necessary for simple algorithms like `tuplesimp` that will appear early in this chapter. On the whole, the advantages seem more important, and we will use this notation throughout.

By repeatedly using Lemma 2.22 (Tuple Intersection) on page 40, it is easy to show that `tuplesimp` is sound:

Fact 2.23 (Tuplesimp Sound) *If* $r \sqsubset t_1 * \ldots * t_h$ *then* `tuplesimp` r *terminates and has the form* $r_1 * \ldots * r_h$*, and* $r \equiv$ `tuplesimp` r.

We can show similar properties for refinements of any ML type constructor, and a similar simplification procedure emerges.

Lemma 2.24 (Refinement Constructor Intersection) *If* $rc_1 \wedge \ldots \wedge rc_n \sqsubset t$ *then*

$$rc_1 \wedge \ldots \wedge rc_n \equiv rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n.$$

Proof: By repeated use of RCON-AND-ELIM-SUB,

$$rc_1 \wedge \ldots \wedge rc_n \leq rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n.$$

To show $rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \leq rc_1 \wedge \ldots \wedge rc_n$, repeatedly use Assumption 2.17 ($\overset{\text{def}}{\wedge}$ Elim) on page 34 to get

$$\text{for all } h \text{ in } 1 \ldots n \text{ we have } rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \overset{\text{def}}{\leq} rc_h.$$

Then RCON-SUB gives

$$\text{for all } h \text{ in } 1 \ldots n \text{ we have } rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \leq rc_h$$

and repeated use of AND-INTRO-SUB gives

$$rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \leq rc_1 \wedge \ldots \wedge rc_n.$$

The first and last displayed formulae imply our conclusions. □

Just as we did with `tuplesimp`, we can define a function that simplifies refinement types that is justified by Lemma 2.24 (Refinement Constructor Intersection) on page 41.

$$\texttt{fun rconsimp } (rc_1 \wedge \ldots \wedge rc_n) = rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n$$

Soundness of this follows from one use of Lemma 2.24 (Refinement Constructor Intersection) on page 41:

Fact 2.25 (Rconsimp Sound) *If* $r \sqsubset tc$ *then* `rconsimp` r *terminates and has the form rc, and* `rconsimp` $r \equiv r$.

TUPLE-SUB tells us that one product refinement type is a subtype another if corresponding components are subtypes. It turns out that the converse is also true, although to prove it we must first strengthen the induction hypothesis as shown in the following theorem.

After we introduce polymorphic refinement type constructors, this will be a trivial consequence of properties of the i operator that we use to prove that each refinement type has finitely many distinct refinements; until then, we need a direct proof.

Lemma 2.26 (Tuple Subtyping) *If*

$$r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn} \leq k_{11} * \ldots * k_{h1} \wedge \ldots \wedge k_{1m} * \ldots * k_{hm}$$

then for all j *between* 1 *and* h *we have*

$$r_{j1} \wedge \ldots \wedge r_{jn} \leq k_{j1} \wedge \ldots \wedge k_{jm}.$$

Proof: By induction on the derivation of our hypothesis.

Case: SELF-SUB | Then $n = m$ and

$$\text{for } i \text{ in } 1 \ldots n \text{ and } j \text{ in } 1 \ldots h \text{ we have } r_{ji} = k_{ji}.$$

and there is a t such that

$$r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn} \sqsubset t. \tag{2.13}$$

The only way to derive (2.13) is by using AND-REF with the premises

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } r_{1i} * \ldots * r_{hi} \sqsubset t.$$

and this can only be derived by using TUPLE-REF when t has the form $t_1 * \ldots * t_n$ and the premises of TUPLE-REF are

$$\text{for } i \text{ in } 1 \ldots n \text{ and } j \text{ in } 1 \ldots h \text{ we have } r_{ji} \sqsubset t_j.$$

Then AND-REF gives

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \sqsubset t_j$$

and then SELF-SUB gives

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq r_{j1} \wedge \ldots \wedge r_{jn}$$

which is our conclusion.

Case: AND-ELIM-R-SUB Then $n > m$ and

$$r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1m} * \ldots * r_{hm} = k_{11} * \ldots * k_{h1} \wedge \ldots \wedge k_{1m} * \ldots * k_{hm}.$$

Thus

$$\text{for } j \text{ in } 1 \ldots h \text{ and } i \text{ in } 1 \ldots m \text{ we have } r_{ji} = k_{ji}.$$

Thus AND-ELIM-L-SUB and AND-ELIM-R-SUB give

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq k_{j1} \wedge \ldots \wedge k_{jm},$$

which is our conclusion.

Case: AND-ELIM-L-SUB Similar.

Case: AND-INTRO-SUB Then there is an i in $1 \ldots m-1$ such that the premises of AND-INTRO-SUB are

$$r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn} \leq k_{11} * \ldots * k_{h1} \wedge \ldots \wedge k_{1i} * \ldots * k_{hi}$$

and

$$r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn} \leq k_{1(i+1)} * \ldots * k_{h(i+1)} \wedge \ldots \wedge k_{1m} * \ldots * k_{hm}.$$

Two uses of the induction hypothesis give

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq k_{j1} \wedge \ldots \wedge k_{ji}.$$

and

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq k_{j(i+1)} \wedge \ldots \wedge k_{jm}.$$

Combining these with AND-INTRO-SUB gives

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq k_{j1} \wedge \ldots \wedge k_{jm},$$

which is our conclusion.

Case: TRANS-SUB For the duration of this case, we will give $r_{11} * \ldots * r_{h1} \wedge \ldots \wedge r_{1n} * \ldots * r_{hn}$ the name r. There is a p such that the premises of TRANS-SUB are

$$r \leq p$$

and

$$p \leq k_{11} * \ldots * k_{h1} \wedge \ldots \wedge k_{1m} * \ldots * k_{hm}.$$

By Theorem 2.21 (Subtypes Refine) on page 36, there is a t such that both r and p refine t. By the form of r we know that t has the form $t_1 * \ldots * t_h$, so by Fact 2.12 (Tuple Refines) on page 32 we know that p has the form $p_{11} * \ldots * p_{h1} \wedge \ldots \wedge p_{1q} * \ldots * p_{hq}$.

Two uses of the induction hypothesis give

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq p_{j1} \wedge \ldots \wedge p_{jq}$$

and

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } p_{j1} \wedge \ldots \wedge p_{jq} \leq k_{j1} \wedge \ldots \wedge k_{jm}.$$

Then we can use TRANS-SUB to get

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_{j1} \wedge \ldots \wedge r_{jn} \leq k_{j1} \wedge \ldots \wedge k_{jm},$$

which is our conclusion.

Case: ARROW-SUB

Case: ARROW-AND-ELIM-SUB

Case: RCON-SUB

Case: RCON-AND-ELIM-SUB

None of these cases can arise because they are not consistent with the form of our hypothesis.

Case: TUPLE-SUB Then $n = 1$ and $m = 1$ and the premises of TUPLE-SUB are our conclusion.

Case: TUPLE-AND-ELIM-SUB | Then $n = 2$ and $m = 1$ and our hypothesis has the form

$$(p_1 * \ldots * p_h) \wedge (p'_1 * \ldots * p'_h) \leq (p_1 \wedge p'_1) * \ldots * (p_h \wedge p'_h).$$

SELF-SUB gives

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } p_j \wedge p'_j \leq p_j \wedge p'_j,$$

which is our conclusion. □

From this it trivially follows that if a valid subtyping inference looks like it could have been inferred by using TUPLE-SUB, then it can be inferred by using TUPLE-SUB:

Corollary 2.27 (TUPLE-SUB **Inversion)** *If*

$$r_1 * \ldots * r_h \leq k_1 * \ldots * k_h$$

then

$$\textit{for } i \textit{ in } 1 \ldots h \textit{ we have } r_i \leq k_i.$$

Proof: Use Lemma 2.26 (Tuple Subtyping) on page 42 with $n = m = 1$. □

The only use we ever make of Lemma 2.26 (Tuple Subtyping) on page 42 is in the proof of Corollary 2.27 (TUPLE-SUB Inversion) on page 45. It is tempting to conjecture that we could eliminate Tuple Subtyping by using `tuplesimp` to prove TUPLE-SUB Inversion directly. Unfortunately, this does not work. The attempted proof of TUPLE-SUB Inversion has the same shape as the proof of Tuple Subtyping, except many cases vanish because the hypothesis has such a special form. The TRANS-SUB case remains, though, and that is where the proof goes wrong. To get the weaker induction hypothesis to apply, we must first use `tuplesimp` on the premises of TRANS-SUB. But then we have no guarantee that the induction makes progress, since using `tuplesimp` makes the type derivation larger.

Similar reasoning to Lemma 2.26 (Tuple Subtyping) on page 42 gives analogous facts about refinement type constructors:

Fact 2.28 (Refinement Constructor Subtyping) *If*

$$rc_1 \wedge \ldots \wedge rc_n \leq kc_1 \wedge \ldots \wedge kc_m$$

then

$$rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \overset{\text{def}}{\leq} kc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} kc_m.$$

Fact 2.29 (RCON-SUB **Inversion)** *If*

$$rc \leq kc$$

then

$$rc \overset{\text{def}}{\leq} kc.$$

After we introduce polymorphic refinement type constructors, both monomorphic refinement type constructors and tuples will be special cases of polymorphic refinement type constructors. Then we will introduce Corollary 5.13 (Arbitrary Constructor Subtyping) on page 250, which generalizes both Fact 2.29 (RCON-SUB Inversion) on page 45 and Corollary 2.27 (TUPLE-SUB Inversion) on page 45.

2.6.2 Splitting

Under appropriate assumptions about the refinement types of `or` and `not`, we should expect the expression

$$\texttt{fn x:}\mathit{bool}\texttt{ => or (x, not x)}$$

to have the refinement type $\top_{bool} \to tt$. The reasoning that leads to the conclusion that the above function always returns a value of type tt relies upon the assumption that all boolean values have one of the types tt or ff. This section formalizes this assertion as $\top_{bool} \asymp \{tt, \mathit{ff}\}$; this assertion is used in type inference in the SPLIT-TYPE rule in Figure 2.6 on page 2.6.

To see why we need to use the fact that $\top_{bool} \asymp \{tt, \mathit{ff}\}$, suppose we made the assumption false by adding a new constructor `maybe` with refinement type $\mathit{runit} \to \top_{bool}$. What is the best type we could expect from `not` if it is passed an argument with type $\top_{bool}$?

At the type level, the behavior of `not` must be monotone. Since $tt \leq \top_{bool}$ and `not` has the type $tt \to \mathit{ff}$, the type we get from `not` must be at least ff. A similar argument leads to the conclusion that it must be at least tt, so the type must be $\top_{bool}$.

We can repeat this argument for `or` instead of `not` and conclude that if we pass something of type $(\top_{bool} * \top_{bool})$ to `or`, then all we can know about the result is that it has the type $\top_{bool}$. Thus the expression `or (maybe (), not (maybe ()))` has the best type $\top_{bool}$, so as long as we have the `maybe` constructor, we cannot give the expression

$$\texttt{fn x:}\mathit{bool}\texttt{ => or (x, not x)}$$

the type $\top_{bool} \to tt$.

2.6.2.1 Definition of Splitting

Therefore our type system has to reason about when a refinement type can be split into a union of other refinement types. We write the assertion that all values of type r have one of the types in the set s as

$$r \asymp s.$$

For example, we have

$$\top_{bool} \asymp \{tt, ff\}$$

and

$$\top_{bool} * \top_{bool} \asymp \{tt * tt, tt * ff, ff * tt, ff * ff\}.$$

We say the elements of s are *fragments* of r and that r splits up into s.

The definition of the splitting relation $\asymp$ for expressions in general relies upon assumptions about how the refinement type constructors behave. We will need to assume that certain refinement type constructors have splits to show that some refinement types have splits. We write the assumption that rc splits up into constructors in the set sc as

$$rc \stackrel{\text{def}}{\asymp} sc.$$

For instance, starting with the assumption

$$\top_{bool} \stackrel{\text{def}}{\asymp} \{tt, ff\}$$

about the refinement type constructors $\top_{bool}$, tt, and ff, we can reach the insipid conclusion

$$\top_{bool} \asymp \{tt, ff\}$$

about the refinement types $\top_{bool}$, tt, ff. We can also reach more interesting conclusions, such as

$$\top_{bool} * \top_{bool} \asymp \{\top_{bool} * tt, \top_{bool} * ff\}.$$

An important property of $\asymp$ is that if a value has a refinement type r and $r \asymp s$ then the value has some element of s as its type. We will have to postpone proof of this until after we define refinement type inference.

We define the $\asymp$ relation in Figure 2.5.

The RCON-SPLIT rule is self-explanatory; it simply allows us to make use of our assumptions.

TUPLE-SPLIT allows us to split up a tuple if we can split any of its components. SML represents functions that take multiple arguments either as curried functions or as functions that take one argument, which is a tuple. Without this rule, type inference for the curried functions would be much stronger than type inference for the functions that take a tuple as an argument.

The TRANS-SPLIT rule lets us use the other rules multiple times to split up a refinement type. Without this, there would be no clear "best" split of some refinement types; for example, TUPLE-TYPE gives $\top_{bool} * \top_{bool} \asymp \{\top_{bool} * tt, \top_{bool} * ff\}$ and $\top_{bool} * \top_{bool} \asymp \{tt * \top_{bool}, ff * \top_{bool}\}$, and neither of these splits is clearly better than the other. With TRANS-SPLIT, we can use TUPLE-TYPE to split the fragments of either of these splits to get

$$\top_{bool} * \top_{bool} \asymp \{tt * tt, tt * ff, ff * tt, ff * ff\},$$

RCON-SPLIT:
$$\frac{rc \overset{\text{def}}{\asymp} sc}{rc \asymp sc}$$

TUPLE-SPLIT:
$$\frac{k_i \asymp s}{k_1 * \ldots * k_{i-1} * k_i * k_{i+1} * \ldots * k_m \asymp \{k_1 * \ldots * k_{i-1} * p * k_{i+1} * \ldots * k_m \mid p \in s\}}$$

TRANS-SPLIT:
$$\frac{r \asymp s_1 \cup \{k\} \qquad k \asymp s_2}{r \asymp s_1 \cup s_2}$$

EQUIV-SPLIT-L:
$$\frac{r \equiv k \qquad k \asymp s}{r \asymp s}$$

EQUIV-SPLIT-R:
$$\frac{p \equiv k \qquad r \asymp s \cup \{k\}}{r \asymp s \cup \{p\}}$$

ELIM-SPLIT:
$$\frac{r \asymp s \cup \{k, p\} \qquad k \leq p}{r \asymp s \cup \{p\}}$$

SELF-SPLIT:
$$\frac{}{r \asymp \{r\}}$$

Figure 2.5: Definition of Splitting

which is in some sense a better split than either of the two splits given earlier. See Subsubsection 2.6.2.2 for a discussion of principal splits.

EQUIV-SPLIT-L and EQUIV-SPLIT-R ensure that the splitting relation is invariant under equivalence. For example, Lemma 2.43 (Split Intersection) on page 54 allows us to start with the premise

$$\top_{bool} \asymp \{tt, \mathit{ff}\}$$

and use that to conclude

$$\top_{bool} \wedge \mathit{ff} \asymp \{tt \wedge \mathit{ff}, \mathit{ff} \wedge \mathit{ff}\}.$$

Without EQUIV-SPLIT-L, the best we would be able to conclude is that for some type r equivalent to $\top_{bool} \wedge \mathit{ff}$ we have

$$r \asymp \{tt \wedge \mathit{ff}, \mathit{ff} \wedge \mathit{ff}\}.$$

If we had EQUIV-SPLIT-L but not EQUIV-SPLIT-R, the best we could conclude is that for some r_1 equivalent to $tt \wedge \mathit{ff}$ and some r_2 equivalent to $\mathit{ff} \wedge \mathit{ff}$,

$$\top_{bool} \wedge \mathit{ff} \asymp \{r_1, r_2\}.$$

ELIM-SPLIT allows us to eliminate unimportant elements from splits. For example, given $\top_{bool} \asymp \{tt, ff, \bot_{bool}\}$, we can use ELIM-SPLIT and $\bot_{bool} \leq tt$ to infer $\top_{bool} \asymp \{tt, ff\}$. If we read the assertion $\top_{bool} \asymp \{tt, ff, \bot_{bool}\}$ as "All values with type $\top_{bool}$ have one of the types tt, ff, or $\bot_{bool}$," it becomes intuitively clear that $\bot_{bool}$ can be eliminated from the split without losing any information.

If we did not have ELIM-SPLIT, then principal splits would not be unique because the unimportant elements could differ. We could still generate a unique minimal set that contained all the information from all of the splits, but without ELIM-SPLIT, that set would not be a split. Since this set would be usable as a split, the distinction between it and the true splits would be formal but not practical. It seems better to erase the unimportant distinction by keeping the ELIM-SPLIT rule.

SELF-SPLIT ensures that each refinement type has at least one split. This simplifies some of the reasoning to come; in particular, Assumption 2.50 (Split Constructor Consistent) on page 66 is flexible enough only because we have SELF-SPLIT.

Now we can prove several lemmas about how $\asymp$ interacts with $\leq$ and $\sqsubset$. First we will assume that the fragments of a refinement type constructor are smaller than the refinement type constructor itself:

Assumption 2.30 (Split Subtype Consistent) *If* $rc \overset{\text{def}}{\asymp} s$ *and* $kc \in s$ *then* $kc \overset{\text{def}}{\leq} rc$.

A straightforward induction lets us lift Split Subtype Consistent from a statement about "$\overset{\text{def}}{\asymp}$" and "$\overset{\text{def}}{\leq}$" to a statement about "$\asymp$" and "$\leq$":

Theorem 2.31 (Splits Are Subtypes I) *If* $r \asymp s \cup \{k\}$ *and* $r \sqsubset t$ *then* $k \leq r$.

Proof: By induction on the derivation of $r \asymp s \cup \{k\}$.

Case: RCON-SPLIT Then r has the form rc and k has the form kc and the premise of RCON-SPLIT is

$$rc \overset{\text{def}}{\asymp} sc \cup \{kc\}.$$

By Assumption 2.30 (Split Subtype Consistent) on page 49, this implies that $kc \overset{\text{def}}{\leq} rc$. Using RCON-SUB on this gives $kc \leq rc$, which is our conclusion.

Case: TUPLE-SPLIT Then r has the from $k_1 * \ldots * k_{i-1} * k_i * k_{i+1} * \ldots * k_m$ and k has the form $k_1 * \ldots * k_{i-1} * p * k_{i+1} * \ldots * k_m$, and the premise of TUPLE-SPLIT is

$$k_i \asymp s' \cup \{p\}.$$

The only way we could have inferred $r \sqsubset t$ is by using TUPLE-REF where t has the form $t_1 * \ldots * t_m$ and one of the premises of TUPLE-REF is

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } k_j \sqsubset t_j.$$

This is true for $j = i$ as well, so we can use the induction hypothesis to give

$$p \leq k_i.$$

By SELF-SUB,

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } k_j \leq k_j,$$

and TUPLE-SUB gives

$$k_1 * \ldots * k_{i-1} * p * k_{i+1} * \ldots * k_m \leq k_1 * \ldots * k_{i-1} * k_i * k_{i+1} * \ldots * k_m,$$

which is our conclusion.

Case: TRANS-SPLIT Then $s \cup \{k\} = s_1 \cup s_2$ where the premises of TRANS-SPLIT are

$$r \asymp s_1 \cup \{p\}$$

and

$$p \asymp s_2.$$

If $k \in s_1$, then our induction hypothesis gives $k \leq r$, and we are done.

If $k \in s_2$, then we reach our conclusion less directly. Our induction hypothesis gives $p \leq r$, and Theorem 2.21 (Subtypes Refine) on page 36 gives a t such that $p \sqsubset t$. Then we can use our induction hypothesis again to get $k \leq p$, and then TRANS-SUB gives $k \leq r$, which is our conclusion.

Case: EQUIV-SPLIT-L Then the premises of EQUIV-SPLIT-L are $r \equiv p$ and $p \asymp s$. Theorem 2.21 (Subtypes Refine) on page 36 gives $p \sqsubset t$, and our induction hypothesis gives $k \leq p$. TRANS-SUB then gives $k \leq r$, which is our conclusion.

Case: EQUIV-SPLIT-R Then we must have $s \cup \{k\} = s' \cup \{p\}$ where the premises of EQUIV-SPLIT-R are $p \equiv p'$ and $r \asymp s' \cup \{p'\}$.

If $k \in s'$, then $k \in s' \cup \{p'\}$, so we can use our induction hypothesis immediately to get $k \leq r$.

If $k = p$, then our induction hypothesis only gives $p' \leq r$. Since $p \equiv p'$, TRANS-SUB gives $p \leq r$, which is our conclusion.

Case: ELIM-SPLIT Then $s \cup \{k\} = s' \cup \{p\}$ and the premises of ELIM-SPLIT are $r \asymp s' \cup \{p', p\}$ and $p' \leq p$. Since k must be in $s' \cup \{p', p\}$, our induction hypothesis gives $k \leq r$.

Case: SELF-SPLIT Then $k = r$, and SELF-SUB gives our conclusion. □

It is a trivial consequence of this show that $\asymp$ and $\sqsubset$ interact reasonably:

Corollary 2.32 (Split Types Refine I) *If* $r \asymp s$ *and* $k \in s$ *and* $r \sqsubset t$ *then* $k \sqsubset t$.

Proof: Let k in s be given. By Theorem 2.31 (Splits Are Subtypes I) on page 49, $k \leq r$. By Theorem 2.21 (Subtypes Refine) on page 36, $k \sqsubset t$. □

The following fact similar to Theorem 2.31 (Splits Are Subtypes I) on page 49 is provable, but the proof is too similar to the proof of Theorem 2.31 (Splits Are Subtypes I) on page 49 for it to be worthwhile to include it here.

Fact 2.33 (Splits Are Subtypes II) *If* $r \asymp s \cup \{k\}$ *and* $k \sqsubset t$ *then* $k \leq r$.

We have an immediate corollary to Fact 2.33 (Splits Are Subtypes II) on page 51 that is completely analogous to Corollary 2.32 (Split Types Refine I) on page 51:

Fact 2.34 (Split Types Refine II) *If* $r \asymp s$ *and* $k \in s$ *and* $k \sqsubset t$ *then* $r \sqsubset t$.

Refinements of an ML type of the form $t_1 \to t_2$ all have a simple form. If $r \sqsubset t_1 \to t_2$, we can use SELF-SPLIT to infer $r \asymp \{r\}$, and we can use EQUIV-SPLIT-R to replace the element of that split by arbitrarily many equivalent elements. A simple induction on the derivation of $r \asymp s$ for any s tells us nothing more interesting than this can happen. This is important in the SPLIT-TYPE case of Lemma 2.70 (Value Substitution) on page 93.

Fact 2.35 (Splits of Arrows are Simple) *If* $r \sqsubset t_1 \to t_2$ *and* $r \asymp s$ *and* $k \in s$ *then* $r \equiv k$.

It is possible to imagine a refinement type having an empty split. This would be consistent with the intuitive meaning of splitting if there were no way to construct a value having that type; for instance, we might expect $\bot_{bool}$ to split into the empty set. However, allowing empty splits causes type inference to behave strangely; see the discussion of the SPLIT-TYPE rule on page 62. We will outlaw refinement types with empty splits; thus possible splits of $\bot_{bool}$ are $\{\bot_{bool}\}$ and trivial variants of this such as $\{\bot_{bool} \wedge \bot_{bool}\}$. First we outlaw empty splits for refinement type constructors:

Assumption 2.36 (Refinement Constructor Splits are Nonempty) *If* $rc \asymp sc$ *then* sc *is nonempty.*

From this it is easy to show that no refinement type can have an empty split:

Fact 2.37 (Splits are Nonempty) *If* $r \asymp s$ *then* s *is nonempty.*

This could be proved by induction on the derivation of $r \asymp s$.

2.6.2.2 Principal Splits

The definition of the $\asymp$ relation gives infinitely many splits of each refinement type. For example, tt has the splits $\{tt\}$, $\{tt \wedge tt\}$, $\{tt, tt \wedge tt\}$, and infinitely many others. A practical type inference algorithm will only have time to consider a finite number of these. In this Subsubsection we will add an assumption that makes it possible for type inference to consider only one split. This split will be a principal split in the sense we will define below. If we identify two splits when there is a one-to-one correspondence between them where equivalent elements correspond, then any well-formed refinement type has exactly one principal split.

First, we shall make a distinction between splits with fragments that could be eliminated by using ELIM-SPLIT and splits without such fragments. The unnecessary fragments add complexity.

Definition 2.38 (Irredundant Splits) *We say a split s is* redundant *if any two elements of s are comparable. Otherwise we say it is* irredundant.

A given refinement type will have many splits, and some of them are more informative than others. A split is informative because it introduces smaller refinement types into the environment. Thus one irredundant split is more informative than another if the former has types smaller than the types in the latter; to put it formally,

Definition 2.39 (Informative Splits) *Given two splits s_1 and s_2 of r, we say that s_1 is* more informative *than s_2 if each element of s_1 is less than some element of s_2.*

Our goal is to have unique most informative irredundant splits:

Definition 2.40 (Principal Splits) *We say that s is a* principal split *of r if s is an irredundant split of r that is more informative than any other irredundant split of r.*

Once we have one principal split, we need not worry about looking for another because there are no other principal splits that are different in any interesting way.

Theorem 2.41 (Unique Principal Splits) *Given any two principal splits of a well-formed refinement type, there is a one-to-one correspondence between them in which the corresponding refinement types are equivalent.*

Proof: Suppose s and s' are principal splits of r, and p is in s. By symmetry, it is sufficient to find a p' in s' such that $p \equiv p'$.

Since s is more informative than s', there is a p' in s' such that $p \leq p'$. Since s' is more informative than s, there is a p'' in s such that $p' \leq p''$. By TRANS-SUB, these imply $p \leq p''$.

Since s is irredundant, we must have $p = p''$. Using this to rewrite $p' \leq p''$ gives $p' \leq p$. This and $p \leq p'$ imply $p \equiv p'$, which implies our conclusion. □

Generally speaking, refinement types that are not well-formed may not have principal splits. For example, suppose

$$\top_{bool} \asymp \{tt, ff\}$$

and

$$\top_{bool} \asymp \{tt, ff, \bot_{bool}\}.$$

In this case the malformed type $(tt \wedge runit) * \top_{bool}$ would have the splits

$$\{(tt \wedge runit) * tt, (tt \wedge runit) * ff\}$$

and

$$\{(tt \wedge runit) * tt, (tt \wedge runit) * ff, (tt \wedge runit) * \bot_{bool}\},$$

among others. These splits are both irredundant because ill-formed refinement types are incomparable, and for the same reason neither split is more informative than the other. By similar reasoning, no splits of $(tt \wedge runit) * \top_{bool}$ will be redundant or more or less informative than any other splits. Thus we only will be interested in principal types for well-formed refinement types.

If we want to have unique most informative splits, we need to have a split more informative than any two given splits. Thus, if we have splits s_1 and s_2 of a well-formed refinement type r then we need to be able to find an s_3 such that

$$r \asymp s_3$$

and

$$\text{for all } k_3 \in s_3 \text{ there is a } k_1 \in s_1 \text{ such that } k_3 \leq k_1$$

and

$$\text{for all } k_3 \in s_3 \text{ there is a } k_2 \in s_2 \text{ such that } k_3 \leq k_2.$$

This will be true if splitting interacts in a natural way with intersection: whenever $p \asymp s$ and p and p' refine the same ML type, we need to ensure that $p \wedge p' \asymp \{p'' \wedge p' \mid p'' \in s\}$. This is intuitively plausible because if a value is in $p \wedge p'$ is in both p and p'. Since it is in p it must be in some $p'' \in s$, and since it is in both p'' and p' it must be in $p'' \wedge p'$.

This property allows us to construct an s_3 more informative than both s_1 and s_2. Let

$$s_3 = \{k_1 \wedge k_2 \mid k_1 \in s_1 \text{ and } k_2 \in s_2\}.$$

The property mentioned in the previous paragraph guarantees that

$$\text{for all } k_1 \text{ in } s_1 \text{ we have } k_1 \asymp \{k_1 \wedge k_2 \mid k_2 \in s_2\}$$

and then we can repeatedly use TRANS-SPLIT to get

$$r \asymp \{k_1 \wedge k_2 \mid k_1 \in s_1 \text{ and } k_2 \in s_2\},$$

which means that s_3 has the properties we want.

What is the best way to ensure that splitting and intersection interact this way? It is sufficient to assume that predefined splitting and predefined intersection interact this way for refinement type constructors:

Assumption 2.42 (Predefined Split Intersection) *If* $rc \stackrel{\text{def}}{\sqsubset} tc$ *and* $kc \stackrel{\text{def}}{\sqsubset} tc$ *and*

$$rc \stackrel{\text{def}}{\asymp} sc$$

then

$$rc \stackrel{\text{def}}{\wedge} kc \stackrel{\text{def}}{\asymp} \{rc' \stackrel{\text{def}}{\wedge} kc \mid rc' \in sc\}.$$

Now we are in a position to prove that the analogous property holds for refinement types in general:

Lemma 2.43 (Split Intersection) *If* $r \sqsubset t$ *and* $k \sqsubset t$ *and*

$$r \asymp s$$

then

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s\}.$$

Proof: By induction on the derivation of $r \asymp s$.

Case: RCON-SPLIT Then r has the form rc and t has the form tc and s has the form sc and the premise of RCON-SPLIT is

$$rc \stackrel{\text{def}}{\asymp} sc.$$

Since $k \sqsubset tc$, we know that k has the form $kc_1 \wedge \ldots \wedge kc_n$. Let $kc = kc_1 \stackrel{\text{def}}{\wedge} \ldots \stackrel{\text{def}}{\wedge} kc_n$. By Lemma 2.24 (Refinement Constructor Intersection) on page 41,

$$rc \wedge k \equiv rc \stackrel{\text{def}}{\wedge} kc.$$

By Assumption 2.42 (Predefined Split Intersection) on page 54,

$$rc \stackrel{\text{def}}{\wedge} kc \stackrel{\text{def}}{\asymp} \{rc' \stackrel{\text{def}}{\wedge} kc \mid rc' \in sc\}.$$

RCON-SPLIT and EQUIV-SPLIT-L give

$$rc \wedge k \asymp \{rc' \stackrel{\text{def}}{\wedge} kc \mid rc' \in sc\}.$$

By Lemma 2.24 (Refinement Constructor Intersection) on page 41 we know that for $rc' \in sc$ we have $rc' \wedge k \equiv rc' \stackrel{\text{def}}{\wedge} kc$. Thus repeated use of EQUIV-SPLIT-R gives

$$rc \wedge k \asymp \{rc' \wedge k \mid rc' \in sc\}$$

which is our conclusion.

Case: TUPLE-SPLIT Then r has the form $r_1 * \ldots * r_n$ and there is an i such that the premise of TUPLE-SPLIT is $r_i \asymp s'$ where

$$s = \{r_1 * \ldots * r_{i-1} * r'' * r_{i+1} * \ldots * r_n \mid r'' \in s'\}.$$

Since $r \sqsubset t$, we know that t has the form $t_1 * \ldots * t_n$. Since $k \sqsubset t$, we know that k has a form that allows us to repeatedly use Lemma 2.22 (Tuple Intersection) on page 40 to find k_1 through k_n such that

$$k \equiv k_1 * \ldots * k_n.$$

By induction hypothesis,

$$r_i \wedge k_i \asymp \{r'' \wedge k_i \mid r'' \in s'\}$$

and TUPLE-SPLIT gives

$$\begin{array}{l}(r_1 \wedge k_1) * \ldots * (r_n \wedge k_n) \asymp \\ \quad \{(r_1 \wedge k_1) * \ldots * (r_{i-1} \wedge k_{i-1}) * (r'' \wedge k_i) * (r_{i+1} \wedge k_{i+1}) * \ldots * (r_n \wedge k_n) \mid r'' \in s'\}.\end{array}$$

All that remains to do is to simplify this until it looks like our conclusion. Lemma 2.22 (Tuple Intersection) on page 40 gives

$$(r_1 * \ldots * r_n) \wedge (k_1 * \ldots * k_n) \equiv (r_1 \wedge k_1) * \ldots * (r_n \wedge k_n)$$

and trivial reasoning about $\wedge$ then gives

$$r \wedge k \equiv (r_1 \wedge k_1) * \ldots * (r_n \wedge k_n).$$

EQUIV-SPLIT-L gives

$$r \wedge k \asymp \{(r_1 \wedge k_1) * \ldots * (r_{i-1} \wedge k_{i-1}) * (r'' \wedge k_i) * (r_{i+1} \wedge k_{i+1}) * \ldots * (r_n \wedge k_n) \mid r'' \in s'\}.$$

Lemma 2.22 (Tuple Intersection) on page 40 gives for r'' in s' we have

$$\begin{array}{l}(r_1 * \ldots * r_{i-1} * r'' * r_{i+1} * \ldots * r_n) \wedge (k_1 * \ldots * k_n) \equiv \\ \quad (r_1 \wedge k_1) * \ldots * (r_{i-1} \wedge k_{i-1}) * (r'' \wedge k_i) * (r_{i+1} \wedge k_{i+1}) * \ldots * (r_n \wedge k_n)\end{array}$$

Trivial reasoning about $\wedge$ gives

$$\begin{array}{l}(r_1 * \ldots * r_{i-1} * r'' * r_{i+1} * \ldots * r_n) \wedge (k_1 * \ldots * k_n) \equiv \\ \quad (r_1 * \ldots * r_{i-1} * r'' * r_{i+1} * \ldots * r_n) \wedge k\end{array}$$

so for r'' in s' we have

$$\begin{array}{l}(r_1 * \ldots * r_{i-1} * r'' * r_{i+1} * \ldots * r_n) \wedge k \equiv \\ \quad (r_1 \wedge k_1) * \ldots * (r_{i-1} \wedge k_{i-1}) * (r'' \wedge k_i) * (r_{i+1} \wedge k_{i+1}) * \ldots * (r_n \wedge k_n).\end{array}$$

Repeated use of EQUIV-SPLIT-R gives

$$r \wedge k \asymp \{(r_1 * \ldots * r_{i-1} * r'' * r_{i+1} * \ldots * r_n) \wedge k \mid r'' \in s'\}$$

and by simple manipulation and the definition of s this implies

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s\},$$

which is our conclusion.

Case: TRANS-SPLIT Then s has the form $s_1 \cup s_2$ where the premises of TRANS-SPLIT are

$$r \asymp s_1 \cup \{p\}$$

and

$$p \asymp s_2.$$

By induction hypothesis,

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s_1 \cup \{p\}\}$$

and by set theory this is equivalent to

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s_1\} \cup \{p \wedge k\}.$$

By Corollary 2.32 (Split Types Refine I) on page 51, $p \sqsubset t$. By induction hypothesis,

$$p \wedge k \asymp \{r' \wedge k \mid r' \in s_2\}.$$

Then TRANS-SPLIT gives

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s_1 \cup s_2\},$$

which is our conclusion.

Case: EQUIV-SPLIT-L Then the premises of EQUIV-SPLIT-L are

$$r \equiv p$$

and

$$p \asymp s.$$

By Theorem 2.21 (Subtypes Refine) on page 36, $p \sqsubset t$, so we can use the induction hypothesis to get

$$p \wedge k \asymp \{r' \wedge k \mid r' \in s\}.$$

Trivial reasoning about $\wedge$ gives $r \wedge k \equiv p \wedge k$, so we can use EQUIV-SPLIT-L to get

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s\},$$

which is our conclusion.

Case: EQUIV-SPLIT-R Then s has the form $s' \cup p$ where the premises of EQUIV-SPLIT-R are

$$p \equiv p'$$

and

$$r \asymp s' \cup p'.$$

By induction hypothesis,

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s'\} \cup \{p' \wedge k\}.$$

Trivial reasoning about $\wedge$ gives $p' \wedge k \equiv p \wedge k$, so we can use EQUIV-SPLIT-R to get

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s'\} \cup \{p \wedge k\},$$

which is our conclusion.

Case: ELIM-SPLIT Then s has the form $s' \cup \{p\}$ where the premises of ELIM-SPLIT are

$$r \asymp s' \cup \{p', p\}$$

and

$$p' \leq p.$$

By induction hypothesis,

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s'\} \cup \{p \wedge k, p' \wedge k\}.$$

Trivial reasoning about $\wedge$ gives $p' \wedge k \leq p \wedge k$, so ELIM-SPLIT gives

$$r \wedge k \asymp \{r' \wedge k \mid r' \in s'\} \cup \{p \wedge k\},$$

which is our conclusion.

Case: SELF-SPLIT Then $s = \{r\}$, so our conclusion is $r \wedge k \asymp \{r \wedge k\}$, which follows from SELF-SPLIT. □

Just as some splits are more informative than others, some splits have no information at all. For example, the split $tt \asymp \{tt, \bot_{bool}\}$ is useless because the meaning of it is a truism: all values of type tt have one of the types tt or $\bot_{bool}$. In general, if a fragment in a split is equivalent to the type we started with, that split is useless. Formally, we have this definition:

Definition 2.44 *If $r \asymp s$ and there is a $k \in s$ such that $r \equiv k$, then we say that s is a* useless split *of r. Otherwise we say it is* useful.

A simple argument tells us that elements of a principal split cannot themselves have useful splits.

Lemma 2.45 (Principal Split Implies Useless Splitting Fragments)
Fragments of a principal split only have useless splits.

Proof: Suppose that an irredundant split of a well-formed refinement type r is $s \cup \{k\}$, where k has a useful split s'. Then by TRANS-SPLIT, $r \asymp s \cup s'$. By definitions of "informative" and "useful", $s \cup s'$ is more informative than $s \cup \{k\}$. By Theorem 2.31 (Splits Are Subtypes I) on page 49, $s \cup s'$ is irredundant. Thus $s \cup \{k\}$ is not a principal split of r. □

We also have the converse:

Lemma 2.46 (Fragments of Principal Split have Useless Splits) *An irredundant split of a well-formed refinement type is principal if all of its fragments only have useless splits.*

Proof: Suppose s is an irredundant split of a well-formed refinement type r, and $r \asymp s'$ and p is in s. We need to show that there is a p' in s' such that $p \leq p'$.

By assumption, there is a t such that $r \sqsubset t$. By Theorem 2.31 (Splits Are Subtypes I) on page 49, this implies $p \leq r$, which means that $p \equiv r \wedge p$. Lemma 2.43 (Split Intersection) on page 54 gives

$$r \wedge p \asymp \{r' \wedge p \mid r' \in s'\},$$

and then EQUIV-SPLIT-L gives

$$p \asymp \{r' \wedge p \mid r' \in s'\}.$$

By assumption, this split of p is useless. Thus there is a p' in s' such that $p \equiv p' \wedge p$, which implies $p \leq p'$, which is our conclusion. □

We will use these two lemmas to build an algorithm for finding principal splits in Subsection 2.10.2.

2.6.3 Refinement Type Inference

Given the subtype relation described above, we can define refinement type inference. The notation is entirely analogous to the ML case. We write

$$\text{VR} \vdash e : r$$

to mean that if we assume each free variable x in e has the refinement type $\text{VR}(x)$, then e has the refinement type r.

If an expression has a refinement type, then it has an ML type, and the refinement type refines the ML type; this is an informal statement of Theorem 2.54 (Inferred Types Refine) on page 68. Since each expression with an ML type has only one ML type, all refinement types for an expression refine the same ML type. If this is not true in general, then not all terms would have principal refinement types. For example, if e has the refinement types tt and $runit$, then e has no principal type because tt, $runit$, and the malformed type $tt \wedge runit$ are all incomparable.

Inferring refinement types for expressions requires making assumptions about the refinement types of constructors. We write the assumption that the constructor c maps values of refinement type r to values of refinement type rc as

$$c \overset{\text{def}}{:} r \hookrightarrow rc.$$

For example, $\texttt{true} \overset{\text{def}}{:} runit \hookrightarrow tt$. We will describe in detail the properties we assume for the $\overset{\text{def}}{:}$ relation in Subsection 2.6.4 on page 64.

The rules for refinement type inference are in Figure 2.6. They are similar to the rules for ML; we have made only the following changes:

We added the AND-INTRO-TYPE rule for introducing intersections. This allows us to infer one type for a function that describes its behavior for several different inputs. For example, since $\cdot \vdash \texttt{fn x}\!:\!bool\ \texttt{=>}\ \texttt{x} : tt \to tt$ and $\cdot \vdash \texttt{fn x}\!:\!bool\ \texttt{=>}\ \texttt{x} : ff \to ff$, we have $\cdot \vdash \texttt{fn x}\!:\!bool\ \texttt{=>}\ \texttt{x} : tt \to tt \wedge ff \to ff$. We do not need a corresponding AND-ELIM-TYPE rule because we can use WEAKEN-TYPE and either AND-ELIM-L-SUB or AND-ELIM-R-SUB to eliminate components from an intersection type.

The AND-INTRO-TYPE rule does not need to assume that r and k refine the same ML type because Theorem 2.54 (Inferred Types Refine) on page 68 guarantees this.

We have added the WEAKEN-TYPE rule. This rule ensures that if an expression has a type, then it also has any larger type. For example, since $\cdot \vdash \texttt{fn x}\!:\!bool\ \texttt{=>}\ \texttt{x} : tt \to tt$, and $tt \to tt \leq tt \to \top_{bool}$, we can infer $\cdot \vdash \texttt{fn x}\!:\!bool\ \texttt{=>}\ \texttt{x} : tt \to \top_{bool}$.

One would hope that if the environment VR has appropriate types for `not` and `or`, we would be able to infer

$$\text{VR} \vdash \texttt{fn x}\!:\!bool\ \texttt{=>}\ \texttt{or (not x, x)} : \top_{bool} \to tt,$$

since this function looks simple and it does indeed return `true ()` for any input. The SPLIT-TYPE rule allows this. We can derive

$$\text{VR}[x := tt] \vdash \texttt{or (not x, x)} : tt$$

and

$$\text{VR}[x := ff] \vdash \texttt{or (not x, x)} : tt$$

and then combine these with SPLIT-TYPE and $\top_{bool} \asymp \{tt, ff\}$ to get

$$\text{VR}[x := \top_{bool}] \vdash \texttt{or (not x, x)} : tt$$

AND-INTRO-TYPE:
$$\frac{\text{VR} \vdash e : r \qquad \text{VR} \vdash e : k}{\text{VR} \vdash e : r \wedge k}$$

WEAKEN-TYPE:
$$\frac{\text{VR} \vdash e : r \qquad r \leq k}{\text{VR} \vdash e : k}$$

SPLIT-TYPE:
$$\frac{k \asymp s \qquad \text{for all } p \text{ in } s \text{ we have } \text{VR}[x := p] \vdash e : r}{\text{VR}[x := k] \vdash e : r}$$

VAR-TYPE:
$$\frac{\text{VR}(x) = r \qquad r \sqsubset t}{\text{VR} \vdash x : r}$$

ABS-TYPE:
$$\frac{\text{VR}[x := r] \vdash e : k \qquad r \sqsubset t}{\text{VR} \vdash \texttt{fn}\ x : t\ \texttt{=>}\ e : r \to k}$$

APPL-TYPE:
$$\frac{\text{VR} \vdash e_1 : k \to r \qquad \text{VR} \vdash e_2 : k}{\text{VR} \vdash e_1\ e_2 : r}$$

CONSTR-TYPE:
$$\frac{c \overset{\text{def}}{:} r \hookrightarrow rc \qquad \text{VR} \vdash e : r}{\text{VR} \vdash c\ e : rc}$$

CASE-TYPE:
$$\frac{\begin{array}{c}\text{VR} \vdash e_0 : rc \\ r \sqsubset u \\ \text{for all } i \text{ in } 1 \ldots n \text{ and all } k, \text{ if } c_i \overset{\text{def}}{:} k \hookrightarrow rc \text{ then } \text{VR} \vdash e_i : k \to r \\ \text{rtom}(\text{VR}) \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end}\!:\!u) :: u\end{array}}{\text{VR} \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end}\!:\!u) : r}$$

TUPLE-TYPE:
$$\frac{\text{for } i \text{ in } 1 \ldots n \text{ we have } \text{VR} \vdash e_i : r_i}{\text{VR} \vdash (e_1, \ \ldots, \ e_n) : r_1 * \ldots * r_n}$$

ELT-TYPE:
$$\frac{\text{VR} \vdash e : r_1 * \ldots * r_n}{\text{VR} \vdash \texttt{elt\_}m\texttt{\_}n\ e : r_m}$$

FIX-TYPE:
$$\frac{\begin{array}{c}r \sqsubset t_1 \to t_2 \\ \text{VR}[f := r] \vdash (\texttt{fn}\ x : t_1\ \texttt{=>}\ e) : r\end{array}}{\text{VR} \vdash (\texttt{fix}\ f : t_1 \to t_2\ \texttt{=>}\ \texttt{fn}\ x : t_1\ \texttt{=>}\ e) : r}$$

Figure 2.6: Monomorphic Refinement Typing Rules

and then use ABS-TYPE to get

$$\text{VR} \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=> or (not x, x)} : \top_{bool} \to tt,$$

which is what we want.

We could get the same result by deleting the SPLIT-TYPE rule and adding the rule

$$\text{SPLIT-SUB:}\frac{r \asymp \{r_1, \ldots, r_n\} \qquad r \to k \sqsubset t}{r_1 \to k \wedge \ldots \wedge r_n \to k \leq r \to k}$$

to the subtyping relation. In this case, we would again start by deriving

$$\text{VR}[x := tt] \vdash \texttt{or (not x, x)} : tt$$

and

$$\text{VR}[x := \mathit{ff}] \vdash \texttt{or (not x, x)} : tt.$$

Then we would apply ABS-TYPE to each of these to get

$$\text{VR} \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=> or (not x, x)} : tt \to tt$$

and

$$\text{VR} \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=> or (not x, x)} : \mathit{ff} \to tt.$$

Combining these with AND-INTRO-TYPE gives

$$\text{VR} \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=> or (not x, x)} : \mathit{ff} \to tt \wedge tt \to tt,$$

and then WEAKEN-TYPE and $\mathit{ff} \to tt \wedge tt \to tt \leq \top_{bool} \to tt$ (from SPLIT-SUB) give

$$\text{VR} \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=> or (not x, x)} : \top_{bool} \to tt,$$

which is our conclusion.

If have SPLIT-TYPE but not SPLIT-SUB, then the types $\mathit{ff} \to tt \wedge tt \to tt$ and $\top_{bool} \to tt$ are not equivalent, even though all values with one type also have the other type. If instead we have SPLIT-SUB but not SPLIT-TYPE, this anomaly does not happen. In this sense, SPLIT-SUB is cleaner than SPLIT-TYPE. It is an open question whether adding SPLIT-SUB would cause inequivalent types to always have different inhabitants.

However, after we add `let` statements in Chapter 4, SPLIT-TYPE becomes stronger than SPLIT-SUB. For example, if we assume the best type for the expression `y mod 3 = 0` is $\top_{bool}$, we would still like the statement

```
let x = (y mod 3 = 0)
in
   or (not x, x)
end
```

to have the refinement type tt. SPLIT-TYPE can do this, but SPLIT-SUB cannot because there is no subexpression of the `let` statement with an appropriate arrow type.

We could still have a strong system with SPLIT-SUB if we defined `let` statements as macros; for example, the above `let` statement would be an abbreviation for

$$\begin{array}{l}(\texttt{fn x: } bool \texttt{ =>} \\ \quad \texttt{or (not x, x))} \\ \texttt{(y mod 3 = 0)}.\end{array}$$

Taking this approach when the `let` statement introduces polymorphism requires first-class polymorphism, which is beyond the scope of this thesis.

At this point we need to combine all the above considerations into a decision. We will keep SPLIT-TYPE because we want the proofs in this chapter to be a special case of the proofs in Chapter 4. We will omit SPLIT-SUB for brevity, since there is no harm in having inequivalent types with identical inhabitants.

TheSPLIT-TYPE rule leads to at least two problems if we allow empty splits. The first problem is that empty splits can be used to infer a malformed refinement type for an expression. For example, if we suppose that $\bot_{bool} \asymp \{\}$, then we can use SPLIT-TYPE to infer

$$[x := \bot_{bool}] \vdash () : tt \wedge (tt \rightarrow tt).$$

This problem can be fixed by adding a premise $r \sqsubset t$ to the SPLIT-TYPE rule. The revised rule would read

$$\frac{\begin{array}{c}k \asymp s \\ \text{for all } p \text{ in } s \text{ we have } \mathrm{VR}[x := p] \vdash e : r \\ r \sqsubset t \\ \mathrm{rtom}(\mathrm{VR}) \vdash e :: t\end{array}}{\mathrm{VR}[x := k] \vdash e : r.}$$

By explicitly requiring the resulting refinement type to refine the ML type for the expression, we outlaw malformed types.

Another problem with empty splits is more difficult to solve. Empty splits cause variables in the environment that appear nowhere in an expression to affect the type of the expression. For example, still assuming that $\bot_{bool} \asymp \{\}$, we can use SPLIT-TYPE to prove

$$[x := \bot_{bool}] \vdash \texttt{true } () : \mathit{ff}.$$

This conclusion is reasonable in an eager language because there are no values of type $\bot_{bool}$. However, it is strange because if we changed the environment to $[x := \mathit{ff}]$, we would no longer be able to prove `true` $() : \mathit{ff}$. Since we assume in many places that changing types in the environment for unused variables does not affect the refinement type of an expression, this would invalidate many of the proofs below. To make it clear where we assume this, we will state it as a fact now, and make explicit reference to this fact when we assume it is true.

Fact 2.47 (Non-free Variables are Ignored) *If x is not free in e, then* $\mathrm{VR}[x := r] \vdash e : k$ *if and only if* $\mathrm{VR} \vdash e : k$.

Proof of this is by two trivial inductions, one on the derivation of $\mathrm{VR}[x := r] \vdash e : k$ and one on the derivation of $\mathrm{VR} \vdash e : k$. In both cases, we have to use Fact 2.37 (Splits are Nonempty) on page 51 in the case where the root inference is SPLIT-TYPE and the type of x is being split. In logic, the "only if" case of this theorem is called "weakening" and the "if" case is called "strengthening".

VAR-TYPE is analogous to VAR-VALID rule except we add the premise $r \sqsubset t$. This ensures that all types we use from the environment are well formed. We state this formally and sketch the proof in Fact 2.48 (Free Variables Refine) on page 64. If this were not true, for many of the theorems below we would have to add an assumption that all variables in the environment are well formed.

The differences between CASE-TYPE and CASE-VALID have two causes. First, there is always exactly one t and tc such that $c \stackrel{\text{def}}{::} t \hookrightarrow tc$, but in general there may be many r's and rc's such that $c \stackrel{\text{def}}{:} r \hookrightarrow rc$. This causes the added quantification on k in the CASE-TYPE rule.

Second, we do not want to require unreachable cases to have a refinement type. If a case is never reachable, we do not require it to have a refinement type, so it would not necessarily have an ML type unless we explicitly required it to. The last premise of CASE-TYPE requires the case statement as a whole to have an ML type, and by CASE-VALID, this requires the unreachable cases to have ML types.

There is a natural analogy between instantiating a polymorphic ML type and weakening a refinement type, since both operations replace the type by a less informative type. The analogy is not perfect; in particular, although there are infinite sequences of increasingly instantiated polymorphic types, such as

$$\alpha, \beta \to \gamma, (\delta \to \epsilon) \to \gamma, \ldots,$$

straightforward reasoning tells us there are no infinite sequences of increasingly weak refinement types: because the refinement types are increasingly weak, they must be comparable, so Theorem 2.21 (Subtypes Refine) on page 36 tells us they all refine the same ML type; by Theorem 2.90 (Finite Refinements) on page 115, there are only finitely many distinct refinements of any ML type, so the chain must be finite.

Unfortunately, standard notation obscures this analogy. If the refinement type r_2 is weaker than the refinement type r_1, we write $r_1 \leq r_2$. But in [DM82], among other places, if the type scheme σ_2 is an instance of the type scheme σ_1, we write $\sigma_1 > \sigma_2$. We make no use of the instantiation ordering in this thesis, so we are not faced with a choice between internal inconsistency and external inconsistency.

The original Damas-Milner type inference system [DM82] disallows instantiating the type of the recursion variable in a fixed point immediately before using it, and that system

is decidable. The Milner-Mycroft type inference system [Myc84] is a variant that permits instantiating the type of the recursion variables in fixed points, and that change is sufficient to make the type system undecidable [KTU89]. None of these questions arise for the ML type inference we use in this chapter, because there is no polymorphism. But the following question does arise: which of these systems is refinement types analogous to, and how does that affect decidability?

Both the Damas-Milner and the Milner-Mycroft type systems distinguish type schemes (which can be instantiated) from types (which cannot). In the refinement type system, WEAKEN-TYPE can be applied anywhere, so all refinement types are analogous to the type schemes in polymorphic type inference. In particular, the refinement type of the recursion variable in a fixed point can be weakened before it is used. In this sense, refinement type inference is analogous to the Milner-Mycroft system. However, refinement type inference is decidable because there the ML type of the recursion variable is uniquely determined, and this tightly constrains the search.

If an expression has a refinement type, then the variables it uses have well-formed types in the variable environment. This is the refinement type analogue of Fact 2.5 (ML Free Variables Bound) on page 29. To state it formally,

Fact 2.48 (Free Variables Refine) *If* $\mathrm{VR} \vdash e : r$ *and* x *is free in* e*, then there is a* t *such that* $\mathrm{VR}(x) \sqsubset t$.

Proof of this is by induction on the derivation of $\mathrm{VR} \vdash e : r$.

2.6.4 Properties of Constructors

This subsection describes the properties of value constructor that are directly used by refinement type inference. As we mentioned earlier, we say that a constructor maps values of type r to values of type rc by writing

$$c \overset{\text{def}}{:} r \hookrightarrow rc.$$

For example, this assumption about `true` fits its ordinary meaning:

$$\texttt{true} \overset{\text{def}}{:} runit \hookrightarrow tt$$

as does this assumption about `false`:

$$\texttt{false} \overset{\text{def}}{:} runit \hookrightarrow \mathit{ff}.$$

If a constructor has a refinement type, it also has larger refinement types, so these assumptions are also reasonable:

$$\texttt{true} \overset{\text{def}}{:} runit \hookrightarrow \top_{bool}$$
$$\texttt{false} \overset{\text{def}}{:} runit \hookrightarrow \top_{bool}.$$

Thus for any c we may have many r's and rc's such that $c \overset{\text{def}}{:} r \hookrightarrow rc$. This contrasts with the unique t and tc such that $c \overset{\text{def}}{::} t \hookrightarrow tc$.

For another example, we have these types for the bitstring constructor `Zero`:

$$\begin{array}{ll}
\texttt{Zero} \overset{\text{def}}{:} \bot_{bitstr} \hookrightarrow \bot_{bitstr} & \texttt{Zero} \overset{\text{def}}{:} empty \hookrightarrow \top_{bitstr} \\
\texttt{Zero} \overset{\text{def}}{:} \bot_{bitstr} \hookrightarrow empty & \texttt{Zero} \overset{\text{def}}{:} nf \hookrightarrow nf \\
\texttt{Zero} \overset{\text{def}}{:} \bot_{bitstr} \hookrightarrow nf & \texttt{Zero} \overset{\text{def}}{:} nf \hookrightarrow \top_{bitstr} \\
\texttt{Zero} \overset{\text{def}}{:} \bot_{bitstr} \hookrightarrow \top_{bitstr} & \texttt{Zero} \overset{\text{def}}{:} \top_{bitstr} \hookrightarrow \top_{bitstr}
\end{array}$$

Now we will describe the properties $\overset{\text{def}}{:}$ that are used by refinement type inference. We constrain how $\overset{\text{def}}{:}$ interacts with the refines relation "$\sqsubset$",the splitting relation "$\asymp$", and the subtyping relation "$\leq$".

Constructors and Refines First, we need $\overset{\text{def}}{:}$ to be consistent with $\overset{\text{def}}{::}$.

Assumption 2.49 (Constructor Type Refines) *If*

$$c \overset{\text{def}}{:} r \hookrightarrow \mathit{rc}$$

and

$$c \overset{\text{def}}{::} t \hookrightarrow \mathit{tc}$$

then $r \sqsubset t$ *and* $\mathit{rc} \overset{\text{def}}{\sqsubset} \mathit{tc}$.

By Lemma 2.10 (Unique ML Types) on page 31, for each constructor c there are t and tc such that $c \overset{\text{def}}{::} t \hookrightarrow tc$. Therefore Assumption 2.49 (Constructor Type Refines) on page 65 constrains the refinement type of all constructors.

Constructors and Splits The property of $\asymp$ that makes it useful is Theorem 2.69 (Splitting Value Types) on page 89, which says that if a value has a type that splits, then the value has one of the fragment types. Suppose the value has the form $c\ v$ and it has the type rc that splits into $\{rc_1, \ldots, rc_n\}$. The derivation that gives a type to $c\ v$ will first infer a type for v; call this type r, and we will assume that $c \overset{\text{def}}{:} r \hookrightarrow rc$. We need some way to conclude that $c\ v$ has one of the rc_i's as its type.

A natural criterion is to require $c \overset{\text{def}}{:} r \hookrightarrow rc_i$ for some i. This requirement is too strong to deal with many natural examples. For example, if we distinguish even length and odd length lists of booleans with the declarations

$$\begin{array}{l}
\texttt{datatype}\ \mathit{blist}\ \texttt{=}\ \texttt{nil}\ \texttt{|}\ \texttt{cons of}\ \mathit{bool} * \mathit{blist} \\
\texttt{rectype}\ \mathit{bev}\ \texttt{=}\ \texttt{cons}\ (\top_{bool} * \mathit{bod}\ \texttt{|}\ \texttt{nil}\ (\textit{runit}) \\
\quad \texttt{and}\ \mathit{bod}\ \texttt{=}\ \texttt{cons}\ (\top_{bool} * \mathit{bev})
\end{array}$$

we have

$$\texttt{cons} \stackrel{\text{def}}{:} \top_{bool} * \top_{blist} \hookrightarrow \top_{blist}$$

and

$$\top_{blist} \stackrel{\text{def}}{\asymp} \{bev, bod\}$$

but neither

$$\texttt{cons} \stackrel{\text{def}}{:} \top_{bool} * \top_{blist} \hookrightarrow bev$$

nor

$$\texttt{cons} \stackrel{\text{def}}{:} \top_{bool} * \top_{blist} \hookrightarrow bod.$$

Instead, we require for each split of r, there is a split of rc such that c maps each fragment of r to some fragment of rc. This seems to work well for many ordinary examples. In the above example, r is $\top_{bool} * \top_{blist}$ and we have the following:

$$\top_{bool} * \top_{blist} \asymp \{\top_{bool} * bev, \top_{bool} * bod\}$$
$$\texttt{cons} \stackrel{\text{def}}{:} \top_{bool} * bev \hookrightarrow bod$$
$$\texttt{cons} \stackrel{\text{def}}{:} \top_{bool} * bod \hookrightarrow bev$$

This approach only makes sense if a refinement type splits whenever it refines some ML type. For instance, consider the ML datatype

$$\begin{aligned}\texttt{datatype}\ pred\ \texttt{=}\ &\texttt{A of}\ bool \to bool\\ \texttt{|}\ &\texttt{B of}\ bool \to bool\end{aligned}$$

and the refinement type declaration

$$\begin{aligned}\texttt{rectype}\ a\ \texttt{=}\ &\texttt{A}\ (bool \to bool)\\ \texttt{and}\ b\ \texttt{=}\ &\texttt{B}\ (bool \to bool)\end{aligned}$$

The types for the value constructors arising from this are

$$\texttt{A} \stackrel{\text{def}}{:} (\top_{bool} \to \top_{bool}) \hookrightarrow a$$

and

$$\texttt{B} \stackrel{\text{def}}{:} (\top_{bool} \to \top_{bool}) \hookrightarrow b.$$

At this point it seems reasonable to have $\top_{pred} \asymp \{a, b\}$. This would fail our criterion if SELF-SPLIT did not ensure that $\top_{bool} \to \top_{bool}$ splits.

Putting this formally,

Assumption 2.50 (Split Constructor Consistent) *If*

$$c \stackrel{\text{def}}{:} r \hookrightarrow rc$$

and

$$rc \overset{\text{def}}{\asymp} \{rc_1, \ldots, rc_n\}$$

then there is some provable assertion of the form

$$r \asymp \{r_1, \ldots, r_m\}$$

such that for all j *between* 1 *and* m *there is an* i *between* 1 *and* n *such that*

$$c \overset{\text{def}}{:} r_j \hookrightarrow rc_i.$$

Constructors and Intersection We will need

Assumption 2.51 (Constructor And Introduction) *If* $c \overset{\text{def}}{:} r \hookrightarrow rc$ *and* $c \overset{\text{def}}{:} r \hookrightarrow kc$ *then* $c \overset{\text{def}}{:} r \hookrightarrow (rc \overset{\text{def}}{\wedge} kc)$.

Any use of CONSTR-TYPE that uses a $\overset{\text{def}}{:}$ property that only exists because of Assumption 2.51 (Constructor And Introduction) on page 67 could be replaced by two uses of CONSTR-TYPE followed by an AND-INTRO-TYPE and then a WEAKEN-TYPE to convert $rc \wedge kc$ to $rc \overset{\text{def}}{\wedge} kc$. We use Assumption 2.51 (Constructor And Introduction) on page 67 when we do not want the derivation to have WEAKEN-TYPE at the root; this is in the RCON-AND-ELIM-SUB case of Lemma 2.67 (Piecewise Intersection) on page 84.

Constructors and Subtyping We need the assumed types for constructors to be consistent with the subtyping relation on the left and the assumed subtyping relation on the right.

Assumption 2.52 (Constructor Argument Strengthen) *If* $c \overset{\text{def}}{:} r \hookrightarrow rc$ *and* $k \leq r$ *then* $c \overset{\text{def}}{:} k \hookrightarrow rc$.

Assumption 2.53 (Constructor Result Weaken) *If* $c \overset{\text{def}}{:} r \hookrightarrow rc$ *and* $rc \overset{\text{def}}{\leq} kc$, *then* $c \overset{\text{def}}{:} r \hookrightarrow kc$.

Neither of these rules change the set of types that can be inferred using CONSTR-TYPE. Any use of CONSTR-TYPE that uses a $\overset{\text{def}}{:}$ property that exists only because Assumption 2.52 (Constructor Argument Strengthen) on page 67 requires it could be replaced by a use of WEAKEN-TYPE followed by a use of CONSTR-TYPE. Similarly, any use of CONSTR-TYPE that uses a $\overset{\text{def}}{:}$ property that exists only because Assumption 2.53 (Constructor Result Weaken) on page 67 requires it could be replaced by a use of CONSTR-TYPE followed by WEAKEN-TYPE.

However, without these assumptions, the CASE-TYPE rule would have to use ":" where it presently uses "$\overset{\text{def}}{:}$". This would make several of the proofs below much more complex, since the behavior of the constructors in CASE-TYPE would depend on the results of type inference rather than depending simply upon our assumptions.

2.7 Compatibility With ML

In this section describe in what sense refinement type inference is compatible with ML type inference. Under very general conditions, all terms with a refinement type have an ML type. Also, under special conditions corresponding to complete absence of `rectype` statements, all terms with an ML type have a refinement type.

The first property ensures that simple modifications to existing ML compilers will allow them to compile programs that have been checked with refinement types. The second property ensures that substituting a compiler that checks refinement types for one that checks ML types will not disorient naive users or break existing code.

2.7.1 Inferring an ML type Given a Refinement Type

The statement of the theorem we intend to prove here is very straightforward. It uses the rtom function defined on page 32:

Theorem 2.54 (Inferred Types Refine) *If*

$$\text{VR} \vdash e : r$$

then there is a t such that

$$r \sqsubset t$$

and

$$\text{rtom}(\text{VR}) \vdash e :: t.$$

The proof of this is an entirely straightforward induction on the refinement type derivation. Explicit provision had to be made in the CASE-TYPE rule to make the proof succeed. The problem is that ML type inference for `case` statements requires all subterms of the `case` statement to have an ML type, but refinement type inference for `case` statements does not require subterms that are obviously unreachable to have a refinement type.

This arrangement is necessary if we want refinement type inference to formalize simple case-based reasoning humans routinely do when they think about a program. For instance, if we assume that the function `f` is well-behaved when passed `true ()` as an argument but not `false ()`, then we would expect the expression

```
case x of
   true => fn ignored:bool => f x
 | false => fn ignored:bool => true ()
end:bool
```

to be well-behaved whether `x` is `true ()` or `false ()`. We can formalize this reasoning in refinement types by giving `f` a type which specifies no behavior when passed an argument of type ff; one such type for `f` would be $tt \to \mathit{ff}$. Then our assertion is that the `case` statement above should have a refinement type even if we give `x` the type ff. Under these assumptions, the expression `f x` has no refinement type, so the rule for `case` statements must not require unreachable cases to have a refinement type.

If refinement type inference completely ignored the unreachable cases in `case` statements, then we could make terms that have a refinement type but no ML type. For example, if we assume that `x` has the refinement type ff, then the statement

```
case x of
   true => fn ignored:bool => () ()
 | false => fn ignored:bool => true ()
end:bool
```

would have a refinement type but no ML type. To solve this problem, the CASE-TYPE explicitly requires the `case` statement to have an ML type.

There is at least one other way to solve the problem. We could allow some expressions to have a refinement type but no ML type. In that case the best we could do here would be to prove that if an expression has both a refinement type and an ML type, the refinement type refines the ML type. Many of the theorems we prove below would need to have a hypothesis added to ensure that some expression has an ML type. The extra hypotheses would add bulk but no insight, so we shall use the CASE-TYPE rule as it stands.

If we eliminated the ML type after the `end` keyword that determines an ML type for the `case` statement some case statements would have malformed refinement types. For example, the refinement type assigned to a case statement where none of the cases were reachable, such as

```
case (fix f:bool→bool => fn x:bool => f x) () of
   true => fn ():tunit => true ()
 | false => fn ():tunit => false ()
end
```

could be a malformed refinement type such as $tt \wedge (tt \to tt)$. If we took out the premise $r \sqsubset u$ from the CASE-TYPE rule, we could infer this type directly; if we left that premise in, but omitted u from the syntax, then we would still be able to use AND-INTRO-TYPE to infer this malformed refinement type for the `case` statement.

Without further ado, we will prove Theorem 2.54 (Inferred Types Refine) on page 68.

Proof: By induction on the derivation of VR $\vdash e : r$.

Case: AND-INTRO-TYPE Then r has the form $r_1 \wedge r_2$ and the premises of AND-INTRO-TYPE are

$$\text{VR} \vdash e : r_1$$

and

$$\text{VR} \vdash e : r_2.$$

Applying our induction hypothesis to each of these gives t_1 and t_2 such the following hold:

$$r_1 \sqsubset t_1$$
$$\text{rtom}(\text{VR}) \vdash e :: t_1$$
$$r_2 \sqsubset t_2$$
$$\text{rtom}(\text{VR}) \vdash e :: t_2.$$

Lemma 2.4 (Unique Inferred ML Types) on page 27 gives $t_1 = t_2$, so AND-REF gives

$$r_1 \wedge r_2 \sqsubset t_1.$$

This and $\text{rtom}(\text{VR}) \vdash e :: t_1$ are our conclusions.

Case: WEAKEN-TYPE The premises of WEAKEN-TYPE must be

$$\text{VR} \vdash e : k$$

and

$$k \leq r.$$

By induction hypothesis, there is a t such that $k \sqsubset t$ and

$$\text{rtom}(\text{VR}) \vdash e :: t.$$

By Theorem 2.21 (Subtypes Refine) on page 36,

$$r \sqsubset t.$$

The last two are our conclusion

Case: SPLIT-TYPE Then VR must have the form $\text{VR}'[x := k]$ where the premises of SPLIT-TYPE are

$$k \asymp s$$

and

$$\text{for all } p \text{ in } s \text{ we have } \text{VR}'[x := p] \vdash e : r.$$

By Fact 2.37 (Splits are Nonempty) on page 51, s is nonempty; let p be any element of s. By induction hypothesis,

$$r \sqsubset t$$

and

$$\text{rtom}(\text{VR}'[x := p]) \vdash e :: t.$$

If $\text{rtom}(k)$ is defined, then Corollary 2.32 (Split Types Refine I) on page 51 gives $\text{rtom}(k) = \text{rtom}(p)$ so we have

$$\text{rtom}(\text{VR}'[x := k]) \vdash e :: t.$$

This and $r \sqsubset t$ are our conclusions.

If rtom(k) is undefined, then by a contrapositive of Fact 2.34 (Split Types Refine II) on page 51, rtom(p) must be undefined also. By definition of rtom applied to functions, this implies rtom(VR$'[x := k]$) = rtom(VR$'[x := p]$). Thus rtom(VR$'[x := k]) \vdash e :: t$; this and $r \sqsubset t$ are our conclusions.

Case: VAR-TYPE Then e has the form x. The premises of VAR-TYPE are $r =$ VR(x) and $r \sqsubset t$. By definition of rtom for functions, rtom(VR)$(x) = t$, so VAR-VALID immediately gives rtom(VR) $\vdash x :: t$, which is our conclusion.

Case: ABS-TYPE Then e has the form `fn` $x : t_1$ `=>` e' and r has the form $r_1 \to r_2$ and the premises of ABS-TYPE are

$$r_1 \sqsubset t_1$$

and

$$\text{VR}[x := r_1] \vdash e' : r_2.$$

Our induction hypothesis gives a t_2 such that

$$r_2 \sqsubset t_2$$

and

$$\text{rtom}(\text{VR}[x := r_1]) \vdash e' :: t_2.$$

Since $r_1 \sqsubset t_1$, we have

$$\text{rtom}(\text{VR}[x := r_1]) = \text{rtom}(\text{VR})[x := t_1]$$

so ABS-VALID gives

$$\text{rtom}(\text{VR}) \vdash \texttt{fn}\ x : t_1\ \texttt{=>}\ e' :: t_1 \to t_2.$$

From $r_1 \sqsubset t_1$ and $r_2 \sqsubset t_2$ we can use ARROW-SUB to get

$$r_1 \to r_2 \sqsubset t_1 \to t_2.$$

The last two displayed formulae are our conclusions.

Case: APPL-TYPE Then e has the form $e_1\ e_2$ and the premises of APPL-TYPE are

$$\text{VR} \vdash e_1 : k \to r$$

and

$$\text{VR} \vdash e_2 : k.$$

Applying induction hypothesis to each of these gives the following:

$$k \to r \sqsubset u$$
$$\text{rtom}(\text{VR}) \vdash e_1 :: u$$
$$k \sqsubset t_1'$$
$$\text{rtom}(\text{VR}) \vdash e_2 :: t_1'$$

The only way to infer $k \to r \sqsubset u$ is by using ARROW-REF where $u = t_1 \to t$ and the premises of ARROW-REF are $k \sqsubset t_1$ and $r \sqsubset t$.

By Lemma 2.10 (Unique ML Types) on page 31, from $k \sqsubset t_1$ and $k \sqsubset t_1'$ we can infer $t_1 = t_1'$. Thus we can use APPL-VALID to get

$$\text{rtom(VR)} \vdash e_1 \ \ e_2 :: t.$$

This and $r \sqsubset t$ are our conclusions.

Case: CONSTR-TYPE Then e has the form $c \ \ e'$ and r has the form rc where the premises of CONSTR-TYPE are

$$c \overset{\text{def}}{:} k \hookrightarrow rc$$

and

$$\text{VR} \vdash e' : k.$$

By Assumption 2.2 (Constructors have Unique ML Types) on page 26, there are unique u and tc such that

$$c \overset{\text{def}}{::} u \hookrightarrow tc.$$

By Assumption 2.49 (Constructor Type Refines) on page 65, $k \sqsubset u$ and $rc \overset{\text{def}}{\sqsubset} tc$.

Our induction hypothesis gives a u' such that $k \sqsubset u'$ and

$$\text{rtom(VR)} \vdash e' :: u'.$$

Since $k \sqsubset u$ and $k \sqsubset u'$, Lemma 2.10 (Unique ML Types) on page 31 tells us that $u = u'$. Thus we can use CONSTR-VALID to get

$$\text{rtom(VR)} \vdash c \ \ e' :: tc.$$

Choose $t = tc$. Since $rc \overset{\text{def}}{\sqsubset} tc$, we can use RCON-REF to get

$$rc \sqsubset tc.$$

The last two displayed equations are our conclusions.

Case: CASE-TYPE

Then e has the form `case` e_0 `of` c_1 `=>` e_1 `|` ... `|` c_n `=>` e_n `end:`t. Two of the premises of CASE-TYPE are

$$r \sqsubset t$$

and

$$\text{rtom(VR)} \vdash e :: t,$$

which are our conclusions.

Case: TUPLE-TYPE Then r has the form $r_1 * \ldots * r_n$ and e has the form $(e_1, \ldots, e_n)$ and the premises of TUPLE-TYPE are

$$\text{for } i \text{ in } 1 \ldots n \text{ we have VR} \vdash e_i : r_i.$$

Applying the induction hypothesis to each of these gives ML types t_1 through t_n such that

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } r_i \sqsubset t_i$$

and

$$\text{for } i \text{ in } 1 \ldots n \text{ we have rtom(VR)} \vdash e_i :: t_i.$$

By TUPLE-REF,

$$r_1 * \ldots * r_n \sqsubset t_1 * \ldots * t_n$$

and by TUPLE-VALID,

$$\text{rtom(VR)} \vdash (e_1 \texttt{ * } \ldots \texttt{ * } e_n) :: t_1 * \ldots * t_n.$$

If we choose $t = t_1 * \ldots * t_n$, this is our conclusion.

Case: ELT-TYPE Then e has the form `elt_`m`_`n e' and the premise of ELT-TYPE is

$$\text{VR} \vdash e' : r_1 * \ldots * r_n$$

where $r = r_m$. By induction hypothesis, there is a u such that

$$r_1 * \ldots * r_n \sqsubset u$$

and

$$\text{rtom(VR)} \vdash e' :: u.$$

We can only infer $r_1 * \ldots * r_n \sqsubset u$ by using TUPLE-REF where u has the form $u_1 * \ldots * u_n$ and

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } r_i \sqsubset u_i.$$

Since u has this form, we can use ELT-VALID to get

$$\text{rtom(VR)} \vdash \texttt{elt\_}m\texttt{\_}n \ e' :: t_m.$$

If we choose $t = t_m$, the last two displayed formulae are our conclusions.

Case: FIX-TYPE Then e has the form `fix` $f : t_1 \to t_2$ `=> fn` $x : t_1$ `=>` e' and the premises of FIX-TYPE are

$$r \sqsubset t_1 \to t_2$$

and

$$\text{VR}[f := r] \vdash \texttt{fn } x : t_1 \texttt{ => } e' : r.$$

By induction hypothesis, there is a t such that

$$r \sqsubset t$$

and

$$\text{rtom}(\text{VR}[f := r]) \vdash \texttt{fn}\ x : t_1\ \texttt{=>}\ e' :: t. \tag{2.14}$$

Since $r \sqsubset t$ and $r \sqsubset t_1 \to t_2$, Lemma 2.10 (Unique ML Types) on page 31 gives $t = t_1 \to t_2$. Since $r \sqsubset t_1 \to t_2$, the definition of rtom gives $\text{rtom}(\text{VR}[f := r]) = \text{rtom}(\text{VR})[f := t_1 \to t_2]$. Thus we can use FIX-VALID on (2.14) to get

$$\text{rtom}(\text{VR}) \vdash \texttt{fix}\ f : t_1 \to t_2\ \texttt{=>}\ \texttt{fn}\ x : t_1\ \texttt{=>}\ e' :: t_1 \to t_2$$

This and $r \sqsubset t_1 \to t_2$ are our conclusions. □

Because of this, if a value has a refinement type, the form of the refinement type gives us information about the form of the value. We will use this in Lemma 2.67 (Piecewise Intersection) on page 84. For example,

Lemma 2.55 (Value Arrow Type) *If* $\text{VR} \vdash v : r_1 \to r_2$ *then* v *has the form* $\texttt{fn}\ x : t\ \texttt{=>}\ e$.

Proof: By Theorem 2.54 (Inferred Types Refine) on page 68, there is a t such that $r_1 \to r_2 \sqsubset t$ and $\text{rtom}(\text{VR}) \vdash v :: t$. Since $r_1 \to r_2 \sqsubset t$, we know t has the form $t_1 \to t_2$. From the ML type inference rules and the possible forms of v, the last inference of $\text{rtom}(\text{VR}) \vdash v :: t$ must be ABS-VALID and v must have the form $\texttt{fn}\ x : t\ \texttt{=>}\ e$. □

Similar reasoning gives results for value constructors and tuples:

Fact 2.56 (Value Constructor Type) *If* $\text{VR} \vdash v : rc$ *then* v *has the form* $c\ v'$.

Fact 2.57 (Value Tuple Type) *If* $\text{VR} \vdash v : r_1 * \ldots * r_n)$ *then* v *has the form* $(v_1, \ldots, v_n)$.

2.7.2 Inferring a Refinement Type Given an ML Type

Some pieces of ML code fail to have a refinement type in the presence of remarkably few `rectype` statements. For example, consider this program in the formal language:

```
datatype d = C of bool → bool
case C (fn x:bool => x) of
   C => fn y:bool → bool => (y (true ()))
end:bool
```

The `case` statement has the ML type $bool$. In the absence of any `rectype` statements, the ML type $bool$ has only one refinement, which we can call $\top_{bool}$. The type of the `case` statement is $\top_{bool}$.

However, if we insert this `rectype` statement before the `datatype` declaration:

$$\texttt{rectype } tt \texttt{ = true } (runit)$$
$$\texttt{ and } ff \texttt{ = false } (runit)$$

the following argument tells us the `case` statement no longer has a refinement type.

Informally, the problem is that the constructor `C` loses all information about its argument. Thus type inference has to make the assumption that y can have any refinement type whatsoever. The worst case is a function that cannot be called legitimately with any value. Since we call y with a value, we fail.

The reader may object at this point that we cannot construct any function that cannot be called with any value. This is true, but in Chapter 6 when we introduce the explicit refinement type declaration operator $\lhd$, we will be able to write such expressions. One of them is:

$$\texttt{fn x:}bool \texttt{ => (x} \lhd \bot_{bool}\texttt{)}.$$

We can also give a formal argument that the `case` statement has no type. Let us suppose that the `case` statement had the refinement type r, and try to construct the type derivation. The conclusion would clearly be

$$\vdash \texttt{case ... end} : r.$$

Since r is arbitrary, we might as well assume the last inference in the derivation is CASE-TYPE rather than AND-INTRO-TYPE or WEAKEN-TYPE. If we use $\top_d$ as the name for the unique refinement of d, the first premise of CASE-TYPE must have the form

$$\vdash \texttt{C (fn x:}bool \texttt{ => x)} : \top_d.$$

The second premise is

$$\top_d \sqsubset d,$$

which is trivial. The third premise says that whenever $C \stackrel{\text{def}}{:} k \hookrightarrow \top_d$, we must have

$$\vdash \texttt{fn y:}bool \to bool \texttt{ => (y (true ()))} : k \to r.$$

Choose $k = \bot_{bool} \to \top_{bool}$. By Lemma 2.68 (Subtype Irrelevancy) on page 88, if we can derive this we can do it with ABS-TYPE as the root inference. The premise of ABS-TYPE must be

$$\texttt{y} : \bot_{bool} \to \top_{bool} \vdash \texttt{y (true ())} : r$$

and this requires using APPL-TYPE with the premise

$$\texttt{y} : \bot_{bool} \to \top_{bool} \vdash \texttt{true ()} : \bot_{bool}$$

which is not derivable.

We can get the `case` statement to typecheck by adding a `rectype` statement so `C` does not lose all information about its argument. One possible addition would be

$$\texttt{rectype } \textit{total} \texttt{ = C } (\top_{bool} \to \top_{bool}).$$

With this addition, there are now two refinements of the ML type d, namely $\top_d$ and $total$. A principal type of `C (fn x:`$bool$ `=> x)` is $total$. (It also has the principal type $total \wedge total$, among infinitely many others; we will eventually show that all principal types are equivalent.) The `case` statement gets the type $\top_{bool}$.

Generalizing from this example, if we allow the programmer to specify any refinement type distinctions, there may be expressions with an ML type but no refinement type. Thus we shall assume for the duration of this subsection that the programmer has made no refinement type distinctions, and we shall prove that any expression with an ML type also has a refinement type. More formally, our temporary assumption is that each ML type constructor tc has exactly one refinement, and we will call that refinement $\overset{\text{def}}{\text{mtor}}(tc)$.

Assumption 2.58 ($\overset{\text{def}}{\text{mtor}}$ **Refines)** *For all tc we have* $\overset{\text{def}}{\text{mtor}}(tc) \overset{\text{def}}{\sqsubset} tc$.

Assumption 2.59 (Only $\overset{\text{def}}{\text{mtor}}$ **Refines)** *(assumed for this subsection only) For all tc and all rc, if* $rc \overset{\text{def}}{\sqsubset} tc$ *then* $rc = \overset{\text{def}}{\text{mtor}}(tc)$.

We then lift this construction in the natural way to refinements of general ML types:

Definition 2.60 *We define* mtor *as the function mapping ML types to refinement types that is consistent with the following equations:*

$$\begin{aligned} \text{mtor}(tc) &= \overset{\text{def}}{\text{mtor}}(tc) \\ \text{mtor}(t_1 \to t_2) &= \text{mtor}(t_1) \to \text{mtor}(t_2) \\ \text{mtor}(t_1 * \ldots * t_n) &= \text{mtor}(t_1) * \ldots * \text{mtor}(t_n). \end{aligned}$$

We extend mtor pointwise to operate on environments such as VM.

Trivial inductions on t give:

Fact 2.61 (mtor **Refines)** *For all* t *we have* $\text{mtor}(t) \sqsubset t$.

Fact 2.62 (Unique Refinement) *If* $r \sqsubset t$ *then* $r \equiv \text{mtor}(t)$.

Now we have enough notation to state that the value constructors behave properly:

Assumption 2.63 (Constructor mtor **Consistent)** *If*

$$c \overset{\text{def}}{::} t \hookrightarrow tc$$

then

$$c \overset{\text{def}}{:} \text{mtor}(t) \hookrightarrow \overset{\text{def}}{\text{mtor}}(tc).$$

In the absence of any of the type declarations introduced in the next chapter, these conditions are satisfied trivially because each ML type constructor has exactly one refinement. Each value constructor maps the unique refinement of its domain to the unique refinement of its range.

Under these assumptions, we can show that each program with an ML type t has the refinement type mtor(t):

Theorem 2.64 (ML Compatibility) *If*

$$\text{VM} \vdash e :: u$$

then

$$\text{mtor}(\text{VM}) \vdash e : \text{mtor}(u).$$

Proof: Straightforward, by induction on the derivation of the hypothesis.

Case: VAR-VALID Then $e = x$ and $u = \text{VM}(x)$. Using VAR-TYPE gives mtor(VM) $\vdash x :$ mtor(u), which is our conclusion.

Case: ABS-VALID Then $e = \texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e'$ and $u = t_1 \to t_2$. The premise of ABS-VALID must be

$$\text{VM}[x := t_1] \vdash e' :: t_2.$$

Fact 2.61 (mtor Refines) on page 76 gives

$$\text{mtor}(t_1) \sqsubset t_1.$$

Our induction hypothesis gives

$$\text{mtor}(\text{VM}[x := t_1]) \vdash e' : \text{mtor}(t_2).$$

Since mtor(VM$[x := t_1]$) = mtor(VM)$[x :=$ mtor$(t_1)]$, we can use ABS-TYPE to get

$$\text{mtor}(\text{VM}) \vdash \texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e' : \text{mtor}(t_1) \to \text{mtor}(t_2).$$

Since mtor$(t_1) \to$ mtor$(t_2) =$ mtor$(t_1 \to t_2)$, this is our conclusion.

Case: APPL-VALID Then $e = e_1\ \ e_2$ and the premises of APPL-VALID are

$$\text{VM} \vdash e_1 :: t \to u$$

and

$$\text{VM} \vdash e_2 :: t.$$

Applying the induction hypothesis to each of these gives

$$\text{mtor}(\text{VM}) \vdash e_1 : \text{mtor}(t \to u)$$

and

$$\text{mtor}(\text{VM}) \vdash e_2 : \text{mtor}(t).$$

Since $\text{mtor}(u \to t) = \text{mtor}(t) \to \text{mtor}(u)$, we can use APPL-TYPE to get

$$\text{mtor}(\text{VM}) \vdash e_1 \;\; e_2 : \text{mtor}(u),$$

which is our conclusion.

Case: CONSTR-VALID Then $e = c \;\; e'$ and $u = tc$ and the premises of CONSTR-VALID are

$$c \overset{\text{def}}{:} t \hookrightarrow tc$$

and

$$\text{VM} \vdash e' :: t.$$

Assumption 2.63 (Constructor mtor Consistent) on page 76 gives

$$c \overset{\text{def}}{::} \text{mtor}(t) \hookrightarrow \overset{\text{def}}{\text{mtor}}(tc)$$

and our induction hypothesis gives

$$\text{mtor}(\text{VM}) \vdash e' : \text{mtor}(t).$$

Using CONSTR-TYPE gives

$$\text{mtor}(\text{VM}) \vdash c \;\; e' : \overset{\text{def}}{\text{mtor}}(tc).$$

Since $\text{mtor}(tc) = \overset{\text{def}}{\text{mtor}}(tc)$, this is our conclusion.

Case: CASE-VALID Then $e = \texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end}\,\texttt{:}\,u$ and the premises of CASE-VALID are

$$\text{VM} \vdash e_0 :: tc,$$

$$\text{for all } i \text{ we have } c_i \overset{\text{def}}{::} t_i \hookrightarrow tc,$$

and

$$\text{for all } i \text{ we have VM} \vdash e_i :: t_i \to u.$$

The induction hypothesis gives the first premise of CASE-TYPE:

$$\text{mtor}(\text{VM}) \vdash e_0 : \text{mtor}(tc)$$

Fact 2.61 (mtor Refines) on page 76 gives the second premise of CASE-TYPE:

$$\text{mtor}(u) \sqsubset u$$

Suppose i and k are given, and that $c_i \overset{\text{def}}{:} k \hookrightarrow \overset{\text{def}}{\text{mtor}}(tc)$. Then Assumption 2.49 (Constructor Type Refines) on page 65, $k \sqsubset t_i$, and Fact 2.62 (Unique Refinement) on page 76 give $k \equiv \text{mtor}(t_i)$. Our induction hypothesis gives

$$\text{mtor}(\text{VM}) \vdash e_i : \text{mtor}(t_i \to u).$$

By the definition of mtor, we have

$$\text{mtor}(\text{VM}) \vdash e_i : \text{mtor}(t_i) \to \text{mtor}(u).$$

By ARROW-SUB and $k \equiv \text{mtor}(t_1)$ we have $k \to \text{mtor}(u) \equiv \text{mtor}(t_i) \to \text{mtor}(u)$. Thus WEAKEN-TYPE gives

$$\text{mtor}(\text{VM}) \vdash e_i : k \to \text{mtor}(u).$$

Since this argument works for any i and k, the third premise of CASE-TYPE holds, so we have our conclusion:

$$\text{mtor}(\text{VM}) \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}\, u) : \text{mtor}(u)$$

Case: TUPLE-VALID Then e has the form $(e_1, \ldots, e_n)$ and u has the form $t_1 * \ldots * t_n$. The premise of TUPLE-VALID must be

$$\text{for all } i \text{ we have } \text{VM} \vdash e_i :: t_i.$$

Our induction hypothesis gives

$$\text{for all } i \text{ we have } \text{mtor}(\text{VM}) \vdash e_i : \text{mtor}(t_i).$$

TUPLE-TYPE then gives

$$\text{mtor}(\text{VM}) \vdash (e_1, \ldots, e_n) : \text{mtor}(t_1) * \ldots * \text{mtor}(t_n)$$

which is our conclusion since $\text{mtor}(t_1) * \ldots * \text{mtor}(t_n) = \text{mtor}(t_1 * \ldots * t_n)$.

Case: ELT-VALID Then e has the form $\texttt{elt\_}m\texttt{\_}n\ \ e'$ and $u = t_m$, where the premise of ELT-VALID is

$$\text{VM} \vdash e' :: t_1 * \ldots * t_n.$$

Our induction hypothesis gives

$$\text{mtor}(\text{VM}) \vdash e' : \text{mtor}(t_1 * \ldots * t_n).$$

Since $\text{mtor}(t_1 * \ldots * t_n) = \text{mtor}(t_1) * \ldots * \text{mtor}(t_n)$, we can use this as the premise to ELT-TYPE to get

$$\text{mtor}(\text{VM}) \vdash \texttt{elt\_}m\texttt{\_}n\ \ e' : \text{mtor}(t_m),$$

which is our conclusion.

Case: FIX-VALID Then e has the form $\texttt{fix}\ \ f : t_1 \to t_2\ \texttt{=>}\ \texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e'$ and u has the form $t_1 \to t_2$. The premise of FIX-VALID is

$$\text{VM}[f := t_1 \to t_2] \vdash (\texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e') :: t_1 \to t_2.$$

Fact 2.61 (mtor Refines) on page 76 gives

$$\text{mtor}(t_1 \to t_2) \sqsubset t_1 \to t_2.$$

Our induction hypothesis gives

$$\text{mtor}(\text{VM}[f := t_1 \to t_2]) \vdash (\texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e') : \text{mtor}(t_1 \to t_2).$$

Since $\text{mtor}(\text{VM}[f := t_1 \to t_2]) = \text{mtor}(\text{VM})[f := \text{mtor}(t_1 \to t_2)]$, we can use FIX-TYPE to get

$$\text{mtor}(\text{VM}) \vdash \texttt{fix}\ \ f : t_1 \to t_2\ \texttt{=>}\ \texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e' : \text{mtor}(\text{VM})$$

which is our conclusion. □

2.8 Simple Soundness Proof

Now we are almost in a position to prove that this type system is sound in an appropriate sense. But first we must prepare the way with some lemmas.

First we will show that as the assumptions in the environment get stronger, the set of types we can infer gets no smaller. At first glance this seems fairly straightforward: Pick a type derivation that we can infer in the weaker environment; to rewrite it to work in the stronger environment, just replace all uses of VAR-TYPE by a use of VAR-TYPE followed by WEAKEN-TYPE, and leave the rest of the derivation the same. Unfortunately, this proof sketch does not handle uses of the SPLIT-TYPE rule in the original derivation. Dealing with SPLIT-TYPE is possible though; see that case of the following proof.

The refinement type inference rules AND-INTRO-TYPE, WEAKEN-TYPE, and SPLIT-TYPE have the same expression in their premise that they have in their conclusion. All of the other refinement type inference rules have smaller expressions in the premises than they have in their conclusion. We say that the latter rules make "syntactic progress". It is often useful to know that the root inference of a derivation of a refinement type for an expression makes syntactic progress because then the form of the expression uniquely determines which refinement type inference rule was used. Therefore we define this special notation to say that the root inference of a type derivation makes syntactic progress:

Definition 2.65 *If* $\text{VR} \vdash e : r$ *and there is a derivation of this that has a rule other than* AND-INTRO-TYPE, WEAKEN-TYPE, *or* SPLIT-TYPE *at the root, then we say* $\text{VR} \Vdash e : r$.

We can think of Lemma 2.66 (Environment Modification) on page 81 as an algorithm that maps a type derivation in the weaker environment to a type derivation in the stronger environment. This algorithm is more useful if the output is a type derivation that makes syntactic progress at the root whenever possible. This is when the original type derivation makes syntactic progress and the expression is not a variable. This optimization is reflected in the theorem by the additional hypotheses and conclusion after the phrase "Also, if in addition".

Lemma 2.66 (Environment Modification) *If*

$$\text{VR} \vdash e : r$$

and

$$\text{VR}' \textit{ has the same domain as } \text{VR}$$

and

$$\textit{for } x \textit{ free in } e \textit{ we have } \text{VR}'(x) \leq \text{VR}(x)$$

then

$$\text{VR}' \vdash e : r.$$

Also, if in addition

$$\text{VR} \Vdash e : r$$

and

$$e \textit{ is not a variable}$$

then

$$\text{VR}' \Vdash e : r.$$

Proof: By induction on the derivation of $\text{VR} \vdash e : r$. Some cases apply only if $\text{VR} \vdash e : r$; other cases apply if either $\text{VR} \vdash e : r$ or $\text{VR} \Vdash e : r$. We will put the cases that apply only if $\text{VR} \vdash e : r$ first.

Case: SPLIT-TYPE Then there is a y such that $\text{VR} = \text{VR}_1[y := k]$ and the premises of SPLIT-TYPE are

$$k \asymp s$$

and

$$\text{for } p \text{ in } s \text{ we have } \text{VR}_1[y := p] \vdash e : r.$$

We take cases on whether y is free in e.

SubCase: y not free in e Thus $\text{VR}' = \text{VR}'_1[y := k']$ where

$$\text{for all } x \text{ free in } e \text{ we have } \text{VR}'_1(x) \leq \text{VR}_1(x).$$

Therefore, trivially,

$$\text{for all } x \text{ free in } e \text{ we have } \text{VR}'_1[y := k](x) \leq \text{VR}_1[y := k](x).$$

Our induction hypothesis gives

$$\text{VR}'_1[y := k] \vdash e : r.$$

Fact 2.47 (Non-free Variables are Ignored) on page 63 gives

$$\text{VR}'_1 \vdash e : r,$$

and then Fact 2.47 (Non-free Variables are Ignored) on page 63 again gives

$$\text{VR}'_1[y := k'] \vdash e : r,$$

which is our conclusion.

SubCase: y free in e Thus $\text{VR}' = \text{VR}'_1[y := k']$ where

$$k' \leq k$$

and

$$\text{for } x \text{ other than } y \text{ free in } e \text{ we have } \text{VR}'_1(x) \leq \text{VR}_1(x).$$

By Lemma 2.43 (Split Intersection) on page 54,

$$k \wedge k' \asymp \{p' \wedge k' \mid p' \in s\}.$$

Since $k' \leq k$, we know that $k \wedge k' \equiv k'$. Thus EQUIV-SPLIT-L gives

$$k' \asymp \{p' \wedge k' \mid p' \in s\}.$$

Since $p' \wedge k'$ is always a subtype of p' when p' is in s, it follows that, for all p' in s and all x free in e,

$$\text{VR}'_1[y := p' \wedge k'](x) \leq \text{VR}_1[y := p'](x).$$

Thus we can use our induction hypothesis to conclude

$$\text{for all } p' \text{ in } s \text{ we have } \text{VR}'_1[y := p' \wedge k'] \vdash e : r$$

Then we can use SPLIT-TYPE to get

$$\text{VR}'_1[y := k'] \vdash e : r,$$

which is our conclusion.

Case: VAR-TYPE Then e has the form x, and $\mathrm{VR}(x) = r$. Since x is free in e, we must have $\mathrm{VR}'(x) \leq r$. VAR-TYPE gives

$$\mathrm{VR}' \vdash x : \mathrm{VR}'(x)$$

and then WEAKEN-TYPE gives

$$\mathrm{VR}' \vdash x : r,$$

which is our conclusion.

Case: AND-INTRO-TYPE

Case: WEAKEN-TYPE

Since neither of these rules use or modify the variable environment, these cases are trivial.

Now we will give the cases that apply when e is not a variable and $\mathrm{VR} \Vdash e : r$.

Case: ABS-TYPE Then e has the form `fn` $x : t_1$ `=>` e' and r has the form $r_1 \to r_2$. The premises of ABS-TYPE must be

$$r_1 \sqsubset t_1$$

and

$$\mathrm{VR}[x := r_1] \vdash e' : r_2.$$

SELF-SUB gives $r_1 \leq r_1$, so we can use our induction hypothesis to get

$$\mathrm{VR}'[x := r_1] \vdash e' : r_2.$$

Then ABS-TYPE gives

$$\mathrm{VR}' \Vdash \texttt{fn}\ x : t_1\ \texttt{=>}\ e' : r_1 \to r_2,$$

which is our conclusion.

Case: FIX-TYPE Then e has the form `fix` $f : t_1 \to t_2$ `=>` `fn` $y : t_1$ `=>` e' and the premises of FIX-TYPE are

$$r \sqsubset t_1 \to t_2$$

and

$$\mathrm{VR}[f := r] \vdash \texttt{fn}\ y : t_1\ \texttt{=>}\ e' : r.$$

SELF-SUB gives $r \leq r$, so we can use the induction hypothesis to get

$$\mathrm{VR}'[f := r] \vdash \texttt{fn}\ y : t_1\ \texttt{=>}\ e' : r$$

and then FIX-TYPE gives

$$\mathrm{VR}' \vdash \texttt{fix}\ f : t_1 \to t_2\ \texttt{=>}\ \texttt{fn}\ y : t_1\ \texttt{=>}\ e' : r,$$

which is our conclusion.

Case: any other inference rule The remaining inference rules do not reference or modify the variable environment, so all remaining cases are trivial. □

We already explicitly take the subtype relation into account in the WEAKEN-TYPE rule. Our next lemma tells us that the subtype relation also describes the behavior of derivations that do not end with WEAKEN-TYPE. This will allow us to eliminate uses of WEAKEN-TYPE from the root of type derivations for values in Lemma 2.68 (Subtype Irrelevancy) on page 88.

Lemma 2.67 (Piecewise Intersection) *If for all i in $1 \ldots n$ we have*

$$\cdot \vdash v : k_i$$

and

$$k_1 \wedge \ldots \wedge k_n \leq r_1 \wedge \ldots \wedge r_m$$

and none of the r_i's or k_i's are themselves intersections of other types, then for all j in $1 \ldots m$ we have

$$\cdot \vdash v : r_j.$$

Proof: By induction on the derivation of $k_1 \wedge \ldots \wedge k_n \leq r_1 \wedge \ldots \wedge r_m$.

Case: SELF-SUB Then our hypothesis is our conclusion.

Case: AND-ELIM-R-SUB Then $n > m$ and for i in $1 \ldots m$ we must have $r_i = k_i$, so our hypothesis immediately implies our conclusion.

Case: AND-ELIM-L-SUB Then $n > m$ and for i in $1 \ldots m$ we have $r_i = k_{n-m+i}$, so once again our hypothesis immediately implies our conclusion.

Case: AND-INTRO-SUB Then there is an h such that the premises of AND-INTRO-SUB are

$$k_1 \wedge \ldots \wedge k_n \leq r_1 \wedge \ldots \wedge r_h$$

and

$$k_1 \wedge \ldots \wedge k_n \leq r_{h+1} \wedge \ldots \wedge r_n.$$

Using the induction hypothesis on each of these gives

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } \cdot \vdash v : r_j$$

and

$$\text{for } j \text{ in } (h+1) \ldots m \text{ we have } \cdot \vdash v : r_j.$$

Together these are our conclusion.

Case: TRANS-SUB Then the premises of TRANS-SUB have the form

$$k_1 \wedge \ldots \wedge k_n \leq p_1 \wedge \ldots \wedge p_q$$

and

$$p_1 \wedge \ldots \wedge p_q \leq r_1 \wedge \ldots \wedge r_m.$$

Using our induction hypothesis on the first of these gives

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } \cdot \Vdash v : p_h$$

and then using it on the second gives

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } \cdot \Vdash v : r_h$$

which is what we wanted to show.

Case: ARROW-SUB Then $n = m = 1$ and k_1 has the form $k \to k'$ and r_1 has the form $r \to r'$. By Lemma 2.55 (Value Arrow Type) on page 74, v has the form `fn` $x : t$ `=>` e. Thus the last inference of $\cdot \Vdash v : k \to k'$ is ABS-TYPE, where the premises of ABS-TYPE are

$$[x := k] \vdash e : k'$$

and

$$k \sqsubset t.$$

Lemma 2.66 (Environment Modification) on page 81 and $r \leq k$ gives

$$[x := r] \vdash e : k'$$

and WEAKEN-TYPE gives

$$[x := r] \vdash e : r'.$$

Then we can use ABS-TYPE to get

$$\cdot \vdash \texttt{fn}\ x : t\ \texttt{=>}\ e : r \to r',$$

which is our conclusion.

Case: ARROW-AND-ELIM-SUB Then $n = 2$ and $m = 1$. The form of ARROW-AND-ELIM-SUB tell us $k_1 \wedge k_2$ has the form $p_1 \to p_2 \wedge p_1 \to p_3$ and r_1 has the form $p_1 \to (p_2 \wedge p_3)$. Our hypothesis tells us

$$\cdot \Vdash v : p_1 \to p_2 \tag{2.15}$$

and

$$\cdot \Vdash v : p_1 \to p_3. \tag{2.16}$$

By Lemma 2.55 (Value Arrow Type) on page 74, v has the form $\texttt{fn}\ x : t\ \texttt{=>}\ e$, so the last inference of both (2.15) and (2.16) must be ABS-TYPE with the following premises:

$$\begin{array}{c} p_1 \sqsubset t \\ [x := p_1] \vdash e : p_2 \\ [x := p_1] \vdash e : p_3 \end{array}$$

Using AND-INTRO-TYPE on the last two of these gives

$$[x := p_1] \vdash e : p_2 \wedge p_3,$$

and then ABS-TYPE gives

$$\cdot \vdash \texttt{fn}\ x : t\ \texttt{=>}\ e : p_1 \rightarrow (p_2 \wedge p_3),$$

which is our conclusion.

Case: RCON-SUB Then $n = m = 1$ and r_1 has the form rc and k_1 has the form kc. The premise of RCON-SUB must be $kc \overset{\text{def}}{\leq} rc$. By Fact 2.56 (Value Constructor Type) on page 74, v has the form $c\ v'$, so the last inference in our hypothesis must be CONSTR-TYPE. The premises of CONSTR-TYPE must be

$$c \overset{\text{def}}{:} r' \hookrightarrow kc$$

and

$$\cdot \vdash v' : r'.$$

Assumption 2.53 (Constructor Result Weaken) on page 67 gives

$$c \overset{\text{def}}{:} r' \hookrightarrow rc$$

and then CONSTR-TYPE gives

$$\cdot \Vdash c\ v' : rc$$

which is our conclusion.

Case: RCON-AND-ELIM-SUB Then $n = 2$ and $m = 1$. The shape of RCON-AND-ELIM-SUB tells us that $k_1 \wedge k_2$ has the form $kc_1 \wedge kc_2$, and r_1 is $kc_1 \overset{\text{def}}{\wedge} kc_2$. By Fact 2.56 (Value Constructor Type) on page 74, v has the form $c\ v'$, so the last inference of the type derivations in our hypothesis must be CONSTR-TYPE. The premises of the uses of CONSTR-TYPE must be

$$\begin{array}{c} c \overset{\text{def}}{:} r'_1 \hookrightarrow kc_1 \\ \cdot \vdash v' : r'_1 \\ c \overset{\text{def}}{:} r'_2 \hookrightarrow kc_2 \\ \cdot \vdash v' : r'_2. \end{array}$$

AND-INTRO-TYPE gives

$$\cdot \vdash v' : r'_1 \wedge r'_2.$$

Two uses of Assumption 2.52 (Constructor Argument Strengthen) on page 67 give

$$c \overset{\text{def}}{:} (r'_1 \wedge r'_2) \hookrightarrow kc_1$$

and

$$c \overset{\text{def}}{:} (r'_1 \wedge r'_2) \hookrightarrow kc_2,$$

and then using Assumption 2.51 (Constructor And Introduction) on page 67 on these gives

$$c \overset{\text{def}}{:} (r'_1 \wedge r'_2) \hookrightarrow (kc_1 \overset{\text{def}}{\wedge} kc_2).$$

Then CONSTR-TYPE gives

$$\cdot \Vdash c\ \ v' : kc_1 \overset{\text{def}}{\wedge} kc_2,$$

which is our conclusion.

Case: TUPLE-SUB Then $m = n = 1$ and k_1 has the form $k'_1 * \ldots * k'_q$ and r_1 has the form $r'_1 * \ldots * r'_q$. The premises of TUPLE-SUB are

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } k'_h \leq r'_h.$$

By Fact 2.57 (Value Tuple Type) on page 74, v has the form $(v_1, \ldots, v_q)$. Thus the last inference of $\cdot \Vdash v : k_1$ must be TUPLE-TYPE and the premises of TUPLE-TYPE are

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } \cdot \vdash v_h : k'_h.$$

Then WEAKEN-TYPE gives

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } \cdot \vdash v_h : r'_h$$

and TUPLE-TYPE gives

$$\cdot \Vdash (v_1,\ \ldots,\ v_q) : r'_1 * \ldots * r'_q,$$

which is our conclusion.

Case: TUPLE-AND-ELIM-SUB Then $n = 2$ and $m = 1$. By the shape of TUPLE-AND-ELIM-SUB, $k_1 \wedge k_2$ must have the form

$$(k'_1 * \ldots * k'_q) \wedge (k''_1 * \ldots * k''_q)$$

and r_1 is

$$(k'_1 \wedge k''_1) * \ldots * (k'_q \wedge k''_q).$$

By Fact 2.57 (Value Tuple Type) on page 74, v must have the form $(v_1,\ \ldots,\ v_q)$ and the last inference of the type derivations in our hypothesis must be TUPLE-TYPE. The premises of the uses of TUPLE-TYPE must be

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } \cdot \vdash v_h : k'_h$$

and

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } \cdot \vdash v_h : k''_h.$$

Using AND-INTRO-TYPE on these gives

$$\text{for } h \text{ in } 1 \ldots q \text{ we have } \cdot \vdash v_h : k'_h \wedge k''_h,$$

and then TUPLE-TYPE gives

$$\cdot \Vdash (v_1\,,\ \ldots,\ v_q) : (k'_1 \wedge k''_1) * \ldots * (k'_q \wedge k''_q),$$

which is our conclusion. □

The previous lemma told us that we can eliminate WEAKEN-TYPE from the root of derivations of a type for a value in an empty environment. The next lemma makes the simple observation that we can eliminate AND-INTRO-TYPE also, if the type is not an intersection. SPLIT-TYPE cannot arise because the environment is empty, so we can always make syntactic progress at the root of a derivation of a non-intersection type for a value.

Lemma 2.68 (Subtype Irrelevancy) *If*

$$\cdot \vdash v : r_1 \wedge \ldots \wedge r_n$$

where none of the r_i's are intersections then for all i in $1 \ldots n$ we have

$$\cdot \Vdash v : r_i.$$

Proof: By induction on the derivation of our hypothesis.

Case: AND-INTRO-TYPE Then there must be an h such that the premises of AND-INTRO-TYPE are

$$\cdot \vdash v : r_1 \wedge \ldots \wedge r_h$$

and

$$\cdot \vdash v : r_{h+1} \wedge \ldots \wedge r_n.$$

Applying the induction hypothesis to the first of these gives, for j in $1 \ldots h$,

$$\cdot \Vdash v : r_j.$$

The induction hypothesis applied to the second of these gives the same for j in $(h+1) \ldots n$, so the two of these are our conclusion.

Case: WEAKEN-TYPE For some r', the premises of WEAKEN-TYPE must be

$$\cdot \vdash v : r'$$

and

$$r' \leq r_1 \wedge \ldots \wedge r_n.$$

Any r' must have the form $r'_1 \wedge \ldots \wedge r'_m$, where none of the r'_j are intersections. By our induction hypothesis, for j in $1 \ldots m$ we have

$$\cdot \Vdash v : r'_j.$$

Then Lemma 2.67 (Piecewise Intersection) on page 84 gives, for j in $1 \ldots n$,

$$\cdot \Vdash v : r_j,$$

which is our conclusion.

Case: SPLIT-TYPE This inference rule requires a nonempty variable environment, which we do not have. Thus this case cannot happen.

Case: VAR-TYPE

Case: APPL-TYPE

Case: CASE-TYPE

Case: ELT-TYPE

Case: FIX-TYPE

All of these rules only apply to non-values, and v is a value. Thus these cases cannot happen.

Case: ABS-TYPE

Case: CONSTR-TYPE

Case: TUPLE-TYPE

In these cases, $n = 1$ and the last inference of our hypothesis is neither AND-INTRO-TYPE nor WEAKEN-TYPE. Thus our hypothesis is

$$\cdot \Vdash v : r_1,$$

which is our conclusion. □

Now we will show that the $\asymp$ relation behaves as one would intuitively expect: If a value has a type that splits, then it has one of the fragments as a type. Formally, we have the following theorem:

Theorem 2.69 (Splitting Value Types) *If $k \asymp s$ and $\cdot \vdash v : k$, then there is an r in s such that $\cdot \vdash v : r$.*

Proof: By induction on the derivation of $k \asymp s$.

Case: RCON-SPLIT Then k has the form kc and all elements of s are refinement type constructors. The only possible form for v is $c\ \ v'$. By Lemma 2.68 (Subtype Irrelevancy) on page 88,

$$\cdot \Vdash c\ \ v' : kc.$$

The last inference of this must be CONSTR-TYPE with the premises

$$c \stackrel{\text{def}}{:} p \hookrightarrow kc$$

and

$$\cdot \vdash v' : p.$$

By Assumption 2.50 (Split Constructor Consistent) on page 66, there is an s' such that $p \asymp s'$, and for all $p' \in s'$ there is a $kc' \in s$ such that $c \stackrel{\text{def}}{:} p' \hookrightarrow kc'$. Since $p \asymp s'$, our induction hypothesis gives a $p' \in s'$ such that

$$\cdot \vdash v' : p'.$$

Let kc' be an element of s such that $c \stackrel{\text{def}}{:} p' \hookrightarrow kc'$. Then CONSTR-TYPE gives

$$\cdot \vdash c\ \ v' : kc',$$

which is our conclusion.

Case: TUPLE-SPLIT Then there must be an h and a q such that k has the form $k_1 * \ldots * k_{h-1} * k_h * k_{h+1} * \ldots * k_q$ and $k_h \asymp s'$ and

$$s = \{k_1 * \ldots * k_{h-1} * p * k_{h+1} * \ldots * k_q \mid p \in s'\}.$$

By Lemma 2.68 (Subtype Irrelevancy) on page 88,

$$\cdot \Vdash v : k_1 * \ldots * k_{h-1} * k_h * k_{h+1} * \ldots * k_q. \tag{2.17}$$

By Fact 2.57 (Value Tuple Type) on page 74, v has the form

$$(v_1,\ \ldots,\ v_{h-1},\ v_h,\ v_{h+1},\ \ldots,\ v_q)$$

and the last inference of (2.17) must be TUPLE-TYPE. The premises of TUPLE-TYPE must be

$$\text{for } j \text{ in } 1 \ldots q \text{ we have } \cdot \vdash v_j : k_j.$$

By induction hypothesis, there is a p is s' such that

$$\cdot \vdash v_h : p.$$

Then TUPLE-TYPE gives

$$\cdot \vdash v : k_1 * \ldots * k_{h-1} * p * k_{h+1} * \ldots * k_q,$$

which is our conclusion.

Case: TRANS-SPLIT Then our hypothesis is $k \asymp s_1 \cup s_2$ where the premises of TRANS-SPLIT are $k \asymp s_1 \cup \{p\}$ and $p \asymp s_2$. By our induction hypothesis, there is a k' in $s_1 \cup \{p\}$ such that

$$\cdot \vdash v : k'.$$

If $k' \in s_1$ this is our conclusion; otherwise $k' = p$ and another use of our induction hypothesis gives a $k'' \in s_2$ such that

$$\cdot \vdash v : k'',$$

which is our conclusion.

Case: EQUIV-SPLIT-L The premises of EQUIV-SPLIT-L must be $k \equiv p$ and $p \asymp s$ for some p. WEAKEN-TYPE gives $\cdot \vdash v : p$, and then our induction hypothesis gives an r in s such that

$$\cdot \vdash v : r,$$

which is our conclusion.

Case: EQUIV-SPLIT-R Then there is a p such that $s = s' \cup \{p\}$ and the premises of EQUIV-SPLIT-R are $p \equiv p'$ and $k \asymp s' \cup \{p'\}$ for some p'. By induction hypothesis, there is some r in $s' \cup \{p'\}$ such that

$$\cdot \vdash v : r.$$

If r is in s' then we are done; otherwise, $r = p'$ and $r \equiv p$. Thus WEAKEN-TYPE gives

$$\cdot \vdash v : p,$$

which is our conclusion.

Case: ELIM-SPLIT Then $s = s' \cup \{p\}$ where the premises of ELIM-SPLIT are $k \asymp s' \cup \{p', p\}$ and $p' \leq p$. By induction hypothesis, there is an r in $s \cup \{p', p\}$ such that

$$\cdot \vdash v : r.$$

If r is in $s \cup \{p\}$, we are done. Otherwise $r = p'$, and WEAKEN-TYPE gives

$$\cdot \vdash v : p,$$

which is our conclusion.

Case: SELF-SPLIT Then $s = \{k\}$, so we can choose $r = k$, so our hypothesis $\cdot \vdash v : k$ is our conclusion. □

We need to establish one more lemma, Lemma 2.70 (Value Substitution) on page 93. This lemma says that substitution for expressions has a natural analogue that works for refinement type derivations. We use this lemma to prove soundness for the semantics rules that use substitution. These rules are APPL-SEM and FIX-SEM, so we only have to be concerned with substituting values or fixed point expressions. It turns out that we cannot do much better than this; in particular, we cannot substitute refinement type derivations for arbitrary expressions into each other.

Since we prove this lemma constructively, the proof of the lemma can be read as an algorithm for doing substitution for derivations. For example, if we perform the substitution

$$[(\texttt{fn y:}\mathit{bool}\ \texttt{=> false ()})/x](\texttt{x (x (true ())))}$$

we get

$$(\texttt{fn y:}\mathit{bool}\ \texttt{=> false ()})\ ((\texttt{fn y:}\mathit{bool}\ \texttt{=> false ()})\ (\texttt{true ()})).$$

Lemma 2.70 (Value Substitution) on page 93 will tell us that, since

$$\cdot \vdash \texttt{fn y:}\mathit{bool}\ \texttt{=> false ()} : \mathit{tt} \to \mathit{ff} \wedge \mathit{ff} \to \mathit{ff} \tag{2.18}$$

and

$$[\texttt{x} := \mathit{tt} \to \mathit{ff} \wedge \mathit{ff} \to \mathit{tt}] \vdash \texttt{x (x (true ()))} : \mathit{ff}, \tag{2.19}$$

we can perform a corresponding substitution on the derivations to get

$$\cdot \vdash (\texttt{fn y:}\mathit{bool}\ \texttt{=> false ()})\ ((\texttt{fn y:}\mathit{bool}\ \texttt{=> false ()})\ (\texttt{true ()})) : \mathit{ff}. \tag{2.20}$$

The strategy for doing this is simple: the constructed derivation has the same shape as the derivation of (2.19), except wherever that derivation examines the type of x, the constructed derivation incorporates a copy of the derivation of (2.18). For example, if we choose this derivation for (2.18):

$$\dfrac{\dfrac{\dfrac{\cdots}{\texttt{y} : \mathit{tt} \vdash \texttt{false ()} : \mathit{ff}}}{\cdot \vdash \texttt{fn y:}\mathit{bool}\ \texttt{=> false ()} : \mathit{tt} \to \mathit{ff}} \quad \dfrac{\dfrac{\cdots}{\texttt{y} : \mathit{ff} \vdash \texttt{false ()} : \mathit{ff}}}{\cdot \vdash \texttt{fn y:}\mathit{bool}\ \texttt{=> false ()} : \mathit{ff} \to \mathit{ff}}}{\cdot \vdash \texttt{fn y:}\mathit{bool}\ \texttt{=> false ()} : \mathit{tt} \to \mathit{ff} \wedge \mathit{ff} \to \mathit{ff}}$$

and this derivation for (2.19) (using r as an abbreviation for $\mathit{tt} \to \mathit{ff} \wedge \mathit{ff} \to \mathit{ff}$):

$$\dfrac{\dfrac{\dfrac{\texttt{x} : r \vdash \texttt{x} : r}{\texttt{x} : r \vdash \texttt{x} : \mathit{ff} \to \mathit{ff}} \quad r \leq \mathit{ff} \to \mathit{ff}}{\texttt{x} : r \vdash \texttt{x} : \mathit{tt} \to \mathit{ff}} \quad \dfrac{\texttt{x} : r \vdash \texttt{x} : r \quad r \leq \mathit{tt} \to \mathit{ff}}{\texttt{x} : r \vdash \texttt{x} : \mathit{tt} \to \mathit{ff}} \quad \dfrac{\cdots}{\texttt{x} : r \vdash \texttt{true ()} : \mathit{tt}}}{x : r \vdash \texttt{x (true ())} : \mathit{ff}}}{x : r \vdash \texttt{x (x (true ()))} : \mathit{ff}}$$

then (where we abbreviate `fn y:`$bool$ `=>` `(false ())` as f and replace all copies of the derivation of (2.18) with "...")

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\cdot \vdash f : r} \quad r \leq \mathit{ff} \to \mathit{ff}}{\cdot \vdash f : \mathit{ff} \to \mathit{ff}} \quad \dfrac{\dfrac{\dfrac{\dots}{\cdot \vdash f : r} \quad r \leq \mathit{tt} \to \mathit{ff}}{\cdot \vdash f : \mathit{tt} \to \mathit{ff}} \quad \dfrac{\dots}{\cdot \vdash \texttt{true}\ () : \mathit{tt}}}{\cdot \vdash f\ (\texttt{true}\ ()) : \mathit{ff}}}{\cdot \vdash f\ ((\texttt{fn y:}\mathit{bool}\ \texttt{=>}\ (\texttt{false}\ ()))\ (\texttt{true}\ ())) : \mathit{ff}}$$

Unlike Fact 2.6 (ML Value Substitution) on page 29, we cannot allow substituting arbitrary expressions in a derivation. For instance, suppose we have a function called `yesno` that asks the user a question to which she can answer yes or no. Our environment VR should assert that `yesno` has the refinement type $\mathit{runit} \to \top_{\mathit{bool}}$. Assuming VR also has appropriate types for `or` and `not`, we can use SPLIT-TYPE to infer

$$\mathrm{VR}[\texttt{x} := \top_{\mathit{bool}}] \vdash \texttt{or}\ (\texttt{not x, x}) : \mathit{tt}$$

and we can infer

$$\mathrm{VR} \vdash \texttt{yesno}\ () : \top_{\mathit{bool}}$$

but doing the substitution to get

$$\mathrm{VR} \vdash \texttt{or}\ (\texttt{not}\ (\texttt{yesno}\ ()), \texttt{yesno}\ ()) : \mathit{tt} \tag{2.21}$$

would lead to unsoundness, since the user could cause the expression to evaluate to `false` by saying "yes" to `yesno ()` the first time and "no" the second time. Even if use the fact that the semantics says the language is completely functional and deterministic so the expression `yesno ()` must either always evaluate to `true` or always evaluate to `false`, the refinement type system cannot infer (2.21). Incidentally, this example shows that refinement types do not rely upon determinacy.

The problem is that the type of `yesno ()` has the split $\{\mathit{tt}, \mathit{ff}\}$, but `yesno ()` has neither of the types tt nor ff. Thus Lemma 2.70 (Value Substitution) on page 93 does not hold for general expressions. Fortunately, it does hold for values and for fixed point expressions, which is all we need.

Lemma 2.70 (Value Substitution) *If* $\mathrm{VR} \vdash e_1 : r_1$, *where* e_1 *is a value or a closed expression of the form* `fix` f`:`t_1 `=>` `fn` x`:`t_2 `=>` e'', *and* $\mathrm{VR}[x := r_1] \vdash e_2 : r_2$, *then* $\mathrm{VR} \vdash [e_1/x]e_2 : r_2$.

Proof: We prove this by induction on the derivation of $\mathrm{VR}[x := r_1] \vdash e_2 : r_2$.

Case: AND-INTRO-TYPE Then r_2 must have the form $r_3 \wedge r_4$ where the premises of AND-INTRO-TYPE are

$$\mathrm{VR}[x := r_1] \vdash e_2 : r_3$$

and

$$\text{VR}[x := r_1] \vdash e_2 : r_4.$$

Applying the induction hypothesis to each of these gives

$$\text{VR} \vdash [e_1/x]e_2 : r_3$$

and

$$\text{VR} \vdash [e_1/x]e_2 : r_4.$$

Using AND-INTRO-TYPE to combine these last two gives

$$\text{VR} \vdash [e_1/x]e_2 : r_3 \wedge r_4$$

which is what we wanted to show.

Case: WEAKEN-TYPE The premises of WEAKEN-TYPE must be

$$\text{VR}[x := r_1] \vdash e_2 : r_3 \tag{2.22}$$

and

$$r_3 \leq r_2. \tag{2.23}$$

Applying the induction hypothesis to (2.22) gives

$$\text{VR} \vdash [e_1/x]e_2 : r_3$$

and applying WEAKEN-TYPE to this and (2.23) gives

$$\text{VR} \vdash [e_1/x]e_2 : r_2$$

which is the desired conclusion.

Case: SPLIT-TYPE Either we are splitting x or some other variable.

SubCase: Splitting type of x Then the premises of SPLIT-TYPE must be

$$r_1 \asymp s$$

and

$$\text{for all } r' \text{ in } s \text{ we have } \text{VR}[x := r'] \vdash e_2 : r_2. \tag{2.24}$$

SubSubCase: e_1 is a closed fixed point By Theorem 2.54 (Inferred Types Refine) on page 68 and FIX-VALID, r_1 must refine an arrow type. Let r' be any element of s; by Fact 2.35 (Splits of Arrows are Simple) on page 51, $r' \equiv r_1$. WEAKEN-TYPE gives

$$\text{VR} \vdash e_1 : r'$$

and then our induction hypothesis gives VR $\vdash [e_1/x]e_2 : r_2$, which is our conclusion.

SubSubCase: e_1 is a value Then Theorem 2.69 (Splitting Value Types) on page 89 tells us that there is an r' in s such that $\cdot \vdash e_1 : r'$. By Fact 2.47 (Non-free Variables are Ignored) on page 63, this implies VR $\vdash e_1 : r'$. Our induction hypothesis applied to this and (2.24) then gives VR $\vdash [e_1/x]e_2 : r_2$, which is our conclusion.

SubCase: Not splitting type of x Then VR has the form VR$'[y := k]$, where we are splitting the type of y. SPLIT-TYPE gives

$$k \asymp s$$

and

$$\text{for } k' \text{ in } s \text{ we have VR}'[y := k', x := r_1] \vdash e_2 : r_2.$$

For each k' in s, our induction hypothesis gives

$$\text{VR}'[y := k'] \vdash [e_1/x]e_2 : r_2.$$

Then SPLIT-TYPE gives

$$\text{VR}'[y := k] \vdash [e_1/x]e_2 : r_2,$$

which is our conclusion.

Case: VAR-TYPE We take subcases depending on whether $e_2 = x$.

SubCase: $e_2 = x$ By VAR-TYPE, $r_1 = r_2$ and $[e_1/x]e_2 = [e_1/x]x = e_1$. Thus the conclusion is one of the hypotheses.

SubCase: $e_2 = y$ and $y \neq x$ Then $[e_1/x]e_2 = e_2$. One of the hypotheses is

$$\text{VR}[x := r_1] \vdash y : r_2.$$

By Fact 2.47 (Non-free Variables are Ignored) on page 63, this implies

$$\text{VR} \vdash y : r_2,$$

which is our conclusion.

Case: ABS-TYPE Then e_2 must have the form `fn` $y : t$ `=>` e'. We take cases on whether $x = y$.

SubCase: $x = y$ Then $[e_1/x]e_2 = [e_1/x]$`fn` $x : t$ `=>` $e' =$ `fn` $x : t$ `=>` $e' = e_2$, so the conclusion is one of the hypotheses.

SubCase: $y \neq x$ From ABS-TYPE we know that r_2 must have the form $r_3 \to r_4$. For some t, the premises of ABS-TYPE must be

$$\text{VR}[x := r_1, y := r_3] \vdash e' : r_4 \tag{2.25}$$

and

$$r_3 \sqsubset t \tag{2.26}$$

Thus

$$[e_1/x]e_2 = \texttt{fn}\ y : t\ \texttt{=>}\ [e_1/x]e'.$$

Applying the induction hypothesis to (2.25) gives

$$\text{VR}[y := r_3] \vdash [e_1/x]e' : r_4$$

and applying ABS-TYPE to this and (2.26) gives

$$\text{VR} \vdash \texttt{fn}\ y : t\ \texttt{=>}\ [e_1/x]e' : r_3 \to r_4$$

which is our conclusion.

Case: APPL-TYPE The conclusion of APPL-TYPE tells us that e_2 has the form $e_3\ \ e_4$ and the premises of APPL-TYPE have the form

$$\text{VR}[x := r_1] \vdash e_3 : r_3 \to r_2$$

and

$$\text{VR}[x := r_1] \vdash e_4 : r_3.$$

Applying the induction hypothesis to each of these gives

$$\text{VR} \vdash [e_1/x]e_3 : r_3 \to r_2$$

and

$$\text{VR} \vdash [e_1/x]e_4 : r_3.$$

Using APPL-TYPE on the last two gives

$$\text{VR} \vdash ([e_1/x]e_3)\ \ ([e_1/x]e_4) : r_2,$$

and since $([e_1/x]e_3)\ \ ([e_1/x]e_4) = [e_1/x](e_3\ \ e_4)$, this is our conclusion.

Case: CONSTR-TYPE Then e_2 has the form $c\ \ e'$ and r_2 has the form rc. The premises of CONSTR-TYPE must be

$$c \overset{\text{def}}{:} r \hookrightarrow rc$$

and

$$\text{VR}[x := r_1] \vdash e' : r.$$

The induction hypothesis gives

$$\text{VR} \vdash [e_1/x]e' : r$$

and then CONSTR-TYPE gives

$$\text{VR} \vdash c\ \ [e_1/x]e' : rc.$$

Since $c\ \ [e_1/x]e' = [e_1/x](c\ \ e')$, this is our conclusion.

Case: CASE-TYPE Then e_2 has the form $\texttt{case}\ e'_0\ \texttt{of}\ c_1\ \texttt{=>}\ e'_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\texttt{=>}\ e'_n\ \texttt{end:}t$ and the premises of CASE-TYPE are

$$\begin{gathered}\text{VR}[x := r_1] \vdash e'_0 : rc,\\ r_2 \sqsubset t,\\ \text{for all } i \text{ in } 1 \ldots n \text{ and all } k, \text{ whenever}\\ c_i \overset{\text{def}}{:} k \hookrightarrow rc \text{ we have}\\ \text{VR}[x := r_1] \vdash e'_i : k \to r_2,\end{gathered} \tag{2.27}$$

and

$$\text{rtom}(\text{VR}[x := r_1]) \vdash e_2 :: t. \tag{2.28}$$

Our induction hypothesis gives

$$\text{VR} \vdash [e_1/x]e'_0 : rc. \tag{2.29}$$

Suppose

$$c_i \overset{\text{def}}{:} k \hookrightarrow rc. \tag{2.30}$$

Then by (2.27), we have

$$\text{VR}[x := r_1] \vdash e'_i : k \to r_2,$$

and our induction hypothesis gives

$$\text{VR} \vdash [e_1/x]e'_i : k \to r_2. \tag{2.31}$$

Theorem 2.54 (Inferred Types Refine) on page 68 and $\text{VR} \vdash e_1 : r_1$ gives

$$\text{rtom}(\text{VR}) \vdash e_1 :: \text{rtom}(r_1).$$

Fact 2.6 (ML Value Substitution) on page 29 applied to this and (2.28) gives

$$\text{rtom}(\text{VR}) \vdash [e_1/x]e_2 :: t \tag{2.32}$$

Now we can use CASE-TYPE on (2.29), $r_2 \sqsubset t$, (2.30) implies (2.31), and (2.32) to get

$$\text{VR} \vdash (\texttt{case}\ [e_1/x]e'_0\ \texttt{of}\ c_1\ \texttt{=>}\ [e_1/x]e'_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ [e_1/x]e'_n\ \texttt{end:}t\ \texttt{:})r_2$$

Since

$$\begin{aligned}&\texttt{case}\ [e_1/x]e'_0\ \texttt{of}\ c_1\ \texttt{=>}\ [e_1/x]e'_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ [e_1/x]e'_n\ \texttt{end:}t =\\ &\quad [e_1/x](\texttt{case}\ e'_0\ \texttt{of}\ c_1\ \texttt{=>}\ e'_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e'_n\ \texttt{end:}t),\end{aligned}$$

this is our conclusion.

Case: TUPLE-TYPE Then e_2 has the form $(e_1,\ \ldots,\ e_n)$ and r_2 has the form $r'_1 * \ldots * r'_n$ and the premises of TUPLE-TYPE must be

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } \mathrm{VR}[x := r_1] \vdash e'_i : r'_i.$$

By induction hypothesis,

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } \mathrm{VR} \vdash [e_1/x]e'_i : r'_i.$$

Then TUPLE-TYPE gives

$$\mathrm{VR} \vdash ([e_1/x]e'_1,\ \ldots,\ [e_1/x]e'_n) : r'_1 * \ldots * r'_n.$$

Since $([e_1/x]e'_1,\ \ldots,\ [e_1/x]e'_n) = [e_1/x](e'_1,\ \ldots,\ e'_n)$ and $r_2 = r'_1 * \ldots * r'_n$, this is our conclusion.

Case: ELT-TYPE Then e_2 has the form $\texttt{elt\_}m\texttt{\_}n\ \ e'$ and the premise of ELT-TYPE must be

$$\mathrm{VR}[x := r_1] \vdash e' : r'_1 * \ldots * r'_n$$

where $r_2 = r'_m$. Our induction hypothesis gives

$$\mathrm{VR} \vdash [e_1/x]e' : r'_1 * \ldots * r'_n$$

and ELT-TYPE then gives

$$\mathrm{VR} \vdash \texttt{elt\_}m\texttt{\_}n\ \ [e_1/x]e' : r'_m.$$

Since $r_2 = r'_m$ and $\texttt{elt\_}m\texttt{\_}n\ \ [e_1/x]e' = [e_1/x]\texttt{elt\_}m\texttt{\_}n\ \ e'$, this is our conclusion.

Case: FIX-TYPE Thus e_2 has the form $\texttt{fix}\ \ f : t_1 \to t_2\ \ \texttt{=>}\ \ \texttt{fn}\ \ y : t_1\ \ \texttt{=>}\ \ e'$. If $x = y$ or $x = f$, then our conclusion is trivial because $[e_1/x]e_2 = e_2$. Otherwise, the premises of FIX-TYPE are

$$r_2 \sqsubset t_1 \to t_2 \tag{2.33}$$

and

$$\mathrm{VR}[f := r_2] \vdash \texttt{fn}\ \ x : t_1\ \ \texttt{=>}\ \ e' : r_2.$$

Our induction hypothesis gives

$$\mathrm{VR}[f := r_2] \vdash [e_1/x]\texttt{fn}\ \ y : t_1\ \ \texttt{=>}\ \ e' : r_2.$$

Since $x \neq y$, this is

$$\mathrm{VR}[f := r_2] \vdash \texttt{fn}\ \ y : t_1\ \ \texttt{=>}\ \ [e_1/x]e' : r_2.$$

Applying FIX-TYPE to this and (2.33) gives

$$\mathrm{VR} \vdash \texttt{fix}\ \ f : t_1 \to t_2\ \ \texttt{=>}\ \ \texttt{fn}\ \ y : t_1\ \ \texttt{=>}\ \ [e_1/x]e' : r_2.$$

Since $x \neq y$ and $x \neq f$, this is our conclusion. □

Now that we have established all of the lemmas we need for the soundness proof, we are in a position to show that this version of the system is sound. So we come to the question: What does it mean for the refinement type system to be sound? It turns out that until we introduce explicit type declarations or references, all expressions that have an ML type will also have a refinement type. Thus the notion of ML type soundness used in Fact 2.3 (ML Type Soundness) on page 27 is trivially true for refinement types, so it is not interesting here. The most interesting thing we can claim at this point is that if we evaluate an expression, the value has the same type as the expression:

Theorem 2.71 (Refinement Type Soundness) *If* $e \Rightarrow v$ *and* $\cdot \vdash e : r$*, then* $\cdot \vdash v : r$.

We could prove this by induction on the derivation of $e \Rightarrow v$. At each step in the proof, there would be three inference rules that could have been used to derive $\cdot \vdash e : r$. They are WEAKEN-TYPE, AND-INTRO-TYPE, and one inference rule that deals specifically with the outermost syntax of e. (SPLIT-TYPE cannot happen here because it requires a nonempty variable environment.) For example, if e has the form $e_1\ e_2$, the last inference rule in the derivation of $e \Rightarrow v$ must be APPL-SEM and the possible inference rules at the root of $\cdot \vdash e_1\ e_2 : r$ are WEAKEN-TYPE, AND-INTRO-TYPE, and APPL-TYPE. The step of the proof dealing with APPL-SEM would have to have another induction on the derivation of $\cdot \vdash e_1\ e_2 : r$ to strip off the outermost uses of WEAKEN-TYPE and AND-INTRO-TYPE, before we could use the outer induction to make more progress on the evaluation trace. Since each step of the proof would have to have this nested induction, the proof would be too large to manage.

It would not work to prove the theorem by induction on the type derivation. The substitution in the APPL-SEM rule can make the expression larger, and therefore it can make the type derivation larger. Thus induction on the type derivation is invalid.

There are two kinds of ways to make progress in the above procedure: We can use AND-INTRO-TYPE or WEAKEN-TYPE to make the type derivation smaller while leaving the evaluation derivation constant, or we can use any of the semantics rules to make the evaluation derivation smaller while possibly making the type derivation larger. All of these possibilities make the ordered pair $(\text{evaluation trace}, \text{type derivation})$ lexicographically smaller. Since any decreasing chain in the lexicographic ordering on the pair is finite, proof by induction on the pair is a valid induction principle, and this is the induction principle we shall use.

The base cases of this induction are the minimal elements in the lexicographic ordering. These consist of a use of a semantics rule that requires no premise paired with a use of a refinement type rule other than WEAKEN-TYPE, AND-INTRO-TYPE, or SPLIT-TYPE. It turns out that there are three base cases:

$$(\text{ABS-SEM}, \text{ABS-TYPE})$$
$$(\text{TUPLE-SEM}, \text{TUPLE-TYPE}) \text{ for a tuple of zero elements}$$
$$(\text{FIX-SEM}, \text{FIX-TYPE})$$

Proof: The proof is by induction on the pair (derivation of $e \Rightarrow v$, derivation of $\cdot \vdash e : r$). We shall label each case with the with an indication of the pairs to which it applies. The label will either be "any" if the case applies to pairs with any value for that component, or the name of an inference rule if the case applies only to pairs where that component of the pair has that inference rule at the root of the derivation. The most interesting case is (APPL-SEM, APPL-TYPE), since that case uses the machinery developed earlier in this chapter. There is an example after this proof on page 103.

Case: (any, AND-INTRO-TYPE) Then r has the form $r_1 \wedge r_2$. The premises of AND-INTRO-TYPE must be $\cdot \vdash e : r_1$ and $\cdot \vdash e : r_2$. Applying the induction hypotheses to each of these gives $\cdot \vdash v : r_1$ and $\cdot \vdash v : r_2$. Combining these with AND-INTRO-TYPE gives

$$\cdot \vdash v : r_1 \wedge r_2$$

which is our conclusion.

Case: (any, WEAKEN-TYPE) The premises of WEAKEN-TYPE must be

$$\cdot \vdash e : r' \tag{2.34}$$

and

$$r' \leq r. \tag{2.35}$$

Applying the induction hypothesis to (2.34) gives

$$\cdot \vdash v : r'.$$

Applying WEAKEN-TYPE to this and (2.35) gives

$$\cdot \vdash v : r,$$

which is our conclusion.

Case: (any, SPLIT-TYPE) This case is unreachable because SPLIT-TYPE assumes the environment is nonempty, but the hypothesis of this theorem assumes it is empty.

Case: (ABS-SEM, ABS-TYPE) By ABS-SEM, $v = e$. Thus our hypothesis $\cdot \vdash e : r$ is our conclusion.

Case: (APPL-SEM, APPL-TYPE) Then e must have the form $e_1\ e_2$ and the premises of APPL-SEM must be

$$e_1 \Rightarrow \texttt{fn}\ x{:}t\ \texttt{=>}\ e_3 \tag{2.36}$$

$$e_2 \Rightarrow v' \tag{2.37}$$

$$[v'/x]e_3 \Rightarrow v \tag{2.38}$$

and the premises of APPL-TYPE must be

$$\cdot \vdash e_1 : r' \to r \tag{2.39}$$
$$\cdot \vdash e_2 : r'. \tag{2.40}$$

Applying the induction hypothesis to (2.36) and (2.39) gives

$$\cdot \vdash \texttt{fn}\ x{:}t\ \texttt{=>}\ e_3 : r' \to r$$

and Lemma 2.68 (Subtype Irrelevancy) on page 88 tells us

$$\cdot \Vdash \texttt{fn}\ x{:}t\ \texttt{=>}\ e_3 : r' \to r.$$

The last inference of this must be ABS-TYPE with the premise

$$x : r' \vdash e_3 : r. \tag{2.41}$$

Applying the induction hypothesis to (2.37) and (2.40) gives

$$\cdot \vdash v' : r'$$

and using Lemma 2.70 (Value Substitution) on page 93 to substitute this into (2.41) gives

$$\cdot \vdash [v'/x]e_3 : r.$$

Applying the induction hypothesis to this and (2.38) gives

$$\cdot \vdash v : r$$

which is our conclusion.

Case: (CONSTR-SEM, CONSTR-TYPE) Then e must have the form $c\ \ e'$ and r must have the form rc and v must have the form $c\ \ v'$ where the premise of CONSTR-SEM is $e' \Rightarrow v'$ and the premises of CONSTR-TYPE are

$$c \overset{\text{def}}{:} k \hookrightarrow rc$$

and

$$\cdot \vdash e' : k.$$

Our induction hypothesis gives

$$\cdot \vdash v' : k$$

and then CONSTR-TYPE gives

$$\cdot \vdash c\ \ v' : rc,$$

which is our conclusion.

Case: (CASE-SEM, CASE-TYPE) The e must have the form

$$\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end}{:}u.$$

The premises of CASE-SEM must be

$$\text{for some } i \text{ we have } e_0 \Rightarrow c_i\ \ v_i$$

and

$$e_i\ \ v_i \Rightarrow v$$

and the premises of CASE-TYPE must include

$$\cdot \vdash e_0 : rc,$$

$$r \sqsubset u,$$

and

$$\begin{array}{l}\text{for all } i \text{ in } 1 \ldots n \text{ and all } k, \text{ if} \\ \qquad c_i \overset{\text{def}}{:} k \hookrightarrow rc \\ \text{then} \\ \qquad \cdot \vdash e_i : k \rightarrow r\end{array} \tag{2.42}$$

Using the induction hypothesis on e_0 gives

$$\cdot \vdash c_i\ \ v_i : rc.$$

Lemma 2.68 (Subtype Irrelevancy) on page 88 then gives

$$\cdot \Vdash c_i\ \ v_i : rc.$$

The last inference of this must be CONSTR-TYPE with the premises

$$c_i \overset{\text{def}}{:} k \hookrightarrow rc$$

and

$$\cdot \vdash v_i : k.$$

By (2.42), we have

$$\cdot \vdash e_i : k \rightarrow r.$$

Using APPL-TYPE on these gives $\cdot \vdash e_i\ \ v_i : r$. Using the induction hypothesis on this gives $\cdot \vdash v : r$, which is our conclusion.

Case: (TUPLE-SEM, TUPLE-TYPE) Then e has the form $(e_1, \ldots, e_n)$ and r has the form $r_1 * \ldots * r_n$ and v has the form $(v_1, \ldots, v_n)$. The premises of TUPLE-SEM are

$$\text{for } i \in 1 \ldots n \text{ we have } e_i \Rightarrow v_i$$

and the premises of TUPLE-TYPE are

$$\text{for } i \in 1 \ldots n \text{ we have } \cdot \vdash e_i : r_i.$$

The induction hypothesis gives

$$\text{for } i \in 1 \ldots n \text{ we have } \cdot \vdash v_i : r_i,$$

and then TUPLE-TYPE gives

$$\cdot \vdash (v_1,\ \ldots,\ v_n) : r_1 * \ldots * r_n,$$

which is our conclusion.

Case: (ELT-SEM, ELT-TYPE) Then e must have the form $\texttt{elt\_}m\texttt{\_}n\ \ e'$. The premise of ELT-SEM must be

$$e' \Rightarrow (v_1,\ \ldots,\ v_n)$$

where $v = v_m$, and the premise of ELT-TYPE must be

$$\cdot \vdash e' : r_1 * \ldots * r_n$$

where $r = r_m$. Our induction hypothesis gives

$$\cdot \vdash (v_1,\ \ldots,\ v_n) : r_1 * \ldots * r_n.$$

By Lemma 2.68 (Subtype Irrelevancy) on page 88, this implies

$$\cdot \nvdash (v_1,\ \ldots,\ v_n) : r_1 * \ldots * r_n.$$

The last inference of this must be TUPLE-TYPE, and one of the premises must be

$$\cdot \vdash v_m : r_m,$$

which is our conclusion.

Case: (FIX-SEM, FIX-TYPE) The e has the form $\texttt{fix}\ \ f : t'\ \texttt{=>}\ \texttt{fn}\ \ x : t''\ \texttt{=>}\ \ e'$. By FIX-TYPE, t' has the form $t_1 \to t_2$ and $t'' = t_1$. The premises of FIX-TYPE must be $r \sqsubset t_1 \to t_2$ and

$$[f := r] \vdash \texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e' : r.$$

Using Lemma 2.70 (Value Substitution) on page 93 on this and

$$\cdot \vdash e : r$$

gives

$$\cdot \vdash [e/f](\texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e') : r \tag{2.43}$$

By FIX-SEM, v is $[e/f](\texttt{fn}\ \ x : t_1\ \texttt{=>}\ \ e')$, so (2.43) is our conclusion. □

The role of SPLIT-TYPE in these theorems is interesting, since it can appear at a non-root position in the type derivation in the hypothesis of Theorem 2.71 (Refinement Type Soundness) on page 99, but none of the cases in that proof deal with that rule. The

resolution to this paradox is that the proof of Lemma 2.70 (Value Substitution) on page 93 never constructs a derivation with SPLIT-TYPE at the root.

This can be most easily understood by walking through the reasoning in the APPL-SEM case of Theorem 2.71 (Refinement Type Soundness) on page 99 for a carefully chosen example. For the purposes of this example, let *or* stand for

```
fn pair:bool * bool =>
   case #1,2 pair of
      true => fn _:tunit => true ()
    | false => fn _:tunit => #2,2 pair
   end:bool
```

and *not* stand for

```
fn b:bool =>
   case b of
      true => fn _:tunit => false ()
    | false => fn _:tunit => true ()
   end:bool.
```

We shall choose $e = (\texttt{fn x:}\mathit{bool}\ \texttt{=>}\ \mathit{or}\ (\mathit{not}\ \texttt{x,}\ \texttt{x}))\ (\texttt{true}\ ())$ and $r = tt$, so we need $v = \texttt{true}\ ()$. We will abbreviate $\mathit{or}\ (\mathit{not}\ \texttt{x,}\ \texttt{x})$ as e' and $\texttt{true}\ ()$ as *tr* when necessary to get the following derivations to fit on a page. The derivation of $e \Rightarrow v$ is

$$\dfrac{\dfrac{}{\texttt{fn x:}\mathit{bool}\ \texttt{=>}\ e' \Rightarrow \texttt{fn x:}\mathit{bool}\ \texttt{=>}\ e'} \quad \dfrac{}{() \Rightarrow ()} \quad \dfrac{\cdots}{\mathit{or}\ (\mathit{not}\ \mathit{tr},\ \mathit{tr}) \Rightarrow \mathit{tr}}}{(\texttt{fn x:}\mathit{bool}\ \texttt{=>}\ e')\ \mathit{tr} \Rightarrow \mathit{tr}}$$

and a derivation of $\cdot \vdash e : r$ is

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{x : \mathit{ff} \vdash e' : tt} \quad \dfrac{\cdots}{x : tt \vdash e' : tt} \quad \top_{bool} \asymp \{tt, \mathit{ff}\}}{x : \top_{bool} \vdash e' : tt}}{\cdot \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=>}\ e' : \top_{bool} \to tt} \quad \dfrac{\dfrac{\cdots}{\cdot \vdash \texttt{true}\ () : tt} \quad tt \leq \top_{bool}}{\cdot \vdash \texttt{true}\ () : \top_{bool}}}{\cdot \vdash (\texttt{fn x:}\mathit{bool}\ \texttt{=>}\ e')\ (\texttt{true}\ ()) : tt.}$$

Notice the use of SPLIT-TYPE in this; it is the rule with the premise $\top_{bool} \asymp \{tt, \mathit{ff}\}$. This example would be less than ideal if we had no size constraint because it is also possible to derive our conclusion without ever using SPLIT-TYPE; we could simply start with $x : tt \vdash e' : tt$, use ABS-TYPE to infer $\cdot \vdash \texttt{fn x:}\mathit{bool}\ \texttt{=>}\ e' : tt \to tt$, and then use APPL-TYPE and $\cdot \vdash \texttt{true}\ () : tt$ to infer our conclusion. For a more serious but larger example, we could replace the $\texttt{true}\ ()$ by an expression with the principal type $\top_{bool}$, thus requiring the use of SPLIT-TYPE to reach the strongest conclusion.

However, let us instead show how the theorem manipulates the example as it stands. First the theorem trivially applies the induction hypothesis to

$$\texttt{fn x}:\mathit{bool}\ \texttt{=>}\ (\mathit{or}\ (\mathit{not}\ \texttt{x},\ \texttt{x})) \Rightarrow \texttt{fn x}:\mathit{bool}\ \texttt{=>}\ (\mathit{or}\ (\mathit{not}\ \texttt{x},\ \texttt{x}))$$

and

$$\cdot \vdash \texttt{fn x}:\mathit{bool}\ \texttt{=>}\ (\mathit{or}\ (\mathit{not}\ \texttt{x},\ \texttt{x})) : \top_{bool} \to tt$$

to get

$$\cdot \vdash \texttt{fn x}:\mathit{bool}\ \texttt{=>}\ (\mathit{or}\ (\mathit{not}\ \texttt{x},\ \texttt{x})) : \top_{bool} \to tt.$$

Then it uses Lemma 2.68 (Subtype Irrelevancy) on page 88 on this to get

$$\cdot \Vdash \texttt{fn x}:\mathit{bool}\ \texttt{=>}\ (\mathit{or}\ (\mathit{not}\ \texttt{x},\ \texttt{x})) : \top_{bool} \to tt,$$

Since the last inference of this must be APPL-TYPE, we must have

$$[x := \top_{bool}] \vdash (\mathit{or}\ (\mathit{not}\ \texttt{x},\ \texttt{x})) : tt. \tag{2.44}$$

Another trivial use of the induction hypothesis uses $\texttt{true}\ () \Rightarrow \texttt{true}\ ()$ and $\cdot \vdash \texttt{true}\ () : \top_{bool}$ to infer

$$\cdot \vdash \texttt{true}\ () : \top_{bool}. \tag{2.45}$$

Then we eliminate the use of SPLIT-TYPE by substituting (2.45) into (2.44) to get

$$\cdot \vdash \mathit{or}\ (\mathit{not}\ (\texttt{true}\ ()),\ (\texttt{true}\ ())) : tt.$$

Using the induction hypothesis on this and $\mathit{or}\ (\mathit{not}\ (\texttt{true}\ ()),\ (\texttt{true}\ ())) \Rightarrow \texttt{true}\ ()$ yields $\cdot \vdash \texttt{true}\ () : tt$, which is our conclusion.

This concludes the soundness proof of the monomorphic version of refinement types. This proof has roughly the same shape as the proofs of soundness for polymorphic refinement types and refinement types with declarations and references.

2.9 Finite Refinements, Principality

Now we shall give several lemmas leading up to the proof that, roughly speaking, each ML type has only finitely many distinct refinements.

This proof below is slightly more complex than necessary. A simpler proof would show by induction on the ML type that each ML type has finitely many distinct refinements. In this proof, the interesting induction case would happen when the ML type has the form $t \to u$; if t has n distinct refinements and u has m distinct refinements, then there are at most $n * m$ distinct refinements of the form $r \to k$ where $r \sqsubset t$ and $k \sqsubset u$. Every refinement of $t \to u$ is equivalent to an intersection of some subset of these, so there are at most 2^{n*m} refinements of $t \to u$.

The problem with this simple approach is that it overestimates the number of refinements of many ML types. For example, the refinement types $\top_{bool} \to tt \wedge tt \to tt$ and $\top_{bool} \to tt$ are equivalent, but they are both counted in the enumeration implied in the argument above. The implementation sometimes has to enumerate the refinements of an ML type, so it is worthwhile to explore a more conservative enumeration in the finiteness proof.

The strategy behind the proof below is to interpret a refinement of $t \to u$ as a monotone function from equivalence classes of t to equivalence classes of u. Two refinement types are equivalent if and only if their interpretations are equal, and we can enumerate without repetition all refinements of a functional ML type by enumerating all monotone functions with an appropriate domain and codomain, as we shall describe below.

For any r refining a functional ML type, we will define the interpretation $I(r)$ of r in terms of a simpler function $i(r)$ that maps refinement types to refinement types instead of equivalence classes to equivalence classes.

There is a natural way to read the interpretation $i(r)$: If f has the type r and x has the type k, then the best type we can infer for $f\ \ x$ is $i(r)(k)$. Our plan is to set up some machinery that allows us to define i in terms of the subtype relation, and then to show that two types k and k' are equivalent if and only if $i(k)$ and $i(k')$ are suitably similar. Then we will define I to be i lifted in a natural way to operate on equivalence classes of refinement types. It will turn out that any types k and k' are equivalent if and only if $I(k)$ and $I(k')$ are equal. Then we finish the proof by showing that there are only finitely many distinct values for $I(k)$.

To make the proof more regular, we will use the symbol ns as the result of $i(r)(k)$ when the corresponding expression would have no type. For example, $i(tt \to ff)(ff) = \mathrm{ns}$. Adding ns requires us to introduce notation for metavariables that can be either a refinement type or ns. We will write these metavariables as $r?$, $k?$ or $p?$ and call the values of these metavariables *generalized refinement types*. Comparing them is straightforward:

Definition 2.72 *We define the binary relation $\preceq$ on generalized refinement types by the following cases:*

$$\begin{array}{rl} r \preceq k & \textit{if and only if } r \leq k \\ r \preceq \mathbf{ns} & \textit{always} \\ \mathbf{ns} \preceq k & \textit{never} \\ \mathbf{ns} \preceq \mathbf{ns}. & \end{array}$$

We can base a natural notion of equivalence on $\preceq$:

Definition 2.73 *We say $r? \approx k?$ if $r? \preceq k?$ and $k? \preceq r?$.*

We could get the same effect by defining $r? \approx k?$ to mean that either $r? \equiv k?$ or $r? = k? = \mathrm{ns}$.

We can also define intersection on generalized refinement types:

Definition 2.74 *We define the binary operation $\triangle$ mapping pairs of generalized refinement types to generalized refinement types by the equations:*

$$r \triangle k = r \wedge k$$
$$r \triangle \mathbf{ns} = \mathbf{ns} \triangle r = r$$
$$\mathbf{ns} \triangle \mathbf{ns} = \mathbf{ns}.$$

The $\triangle$ operation inherits commutativity, associativity, and idempotence from $\wedge$.

The big advantage of $\triangle$ over $\wedge$ is that $\triangle$ has an identity, specifically ns. Thus we can define $\triangle$ to work on a finite set of refinement types even if the set is empty:

Definition 2.75 *If s is a finite set of refinement types, then $\triangle s$ is the following generalized refinement type:*

If s is empty, then $\triangle s = \mathbf{ns}$.

If $s = \{r_1, \ldots, r_n\}$, then $\triangle s = r_1 \wedge \ldots \wedge r_n$.

We shall continue to use s as a finite set of refinement types for the rest of this section.

This definition is slightly ambiguous, since the order of the elements in a set is not determined and $\wedge$ is only commutative if we ignore the difference between equivalent refinement types that are not equal. For example, $\triangle\{tt, \mathit{ff}\}$ could be $tt \wedge \mathit{ff}$ as well as $\mathit{ff} \wedge tt$. This ambiguity makes no difference to the reasoning below, and we shall ignore it.

When all refinement types in s refine the same ML type t, the generalized refinement type $\triangle s$ either refines t or is ns. We extend the notion of refinement to include sets of refinement types and generalized refinement types, so we can simply say that if $s \sqsubset t$, then $\triangle s \sqsubset t$. In this extension of the meaning of $\sqsubset$, both the empty set $\{\}$ and ns both refine all ML types.

The $\triangle$ operator has several properties that follow from analogous properties of $\wedge$, commutativity and associativity of $\wedge$, and trivial induction arguments:

Fact 2.76 ($\triangle$ Elim Sub) *If $s \supset s'$ and $s \sqsubset t$ then*

$$\triangle s \preceq \triangle s'.$$

Fact 2.77 ($\triangle$ Intro Sub) *If s_1, s_2, and s_3 all refine t and*

$$\triangle s_1 \preceq \triangle s_2$$

and

$$\triangle s_1 \preceq \triangle s_3$$

then

$$\triangle s_1 \preceq \triangle(s_2 \cup s_3).$$

Fact 2.78 (Transitivity of $\preceq$) *If $r? \preceq k?$ and $k? \preceq p?$ then $r? \preceq p?$.*

Now we have enough machinery to define $i(r)$ and prove some simple properties of it:

Definition 2.79 *Suppose $k? \sqsubset t \to t'$ and $r \sqsubset t$. If $k?$ has the form $k_1 \to k_1' \wedge \ldots \wedge k_n \to k_n'$, we define*

$$i(k?)(r) = \triangle\{k_j' \mid j \text{ is between } 1 \text{ and } n \text{ and } r \leq k_j\}.$$

Otherwise $k? = \mathbf{ns}$ and we define $i(k?)(r) = \mathbf{ns}$.

For example, if

$$k = tt \to \top_{bool} \wedge \top_{bool} \to \mathit{ff} \wedge \mathit{ff} \to tt$$

then

$$\begin{aligned} i(k)(tt) &= \top_{bool} \wedge \mathit{ff} \equiv \mathit{ff} \\ i(k)(\mathit{ff}) &= \mathit{ff} \wedge tt \equiv \bot_{bool} \\ i(k)(\top_{bool}) &= \mathit{ff} \\ i(k)(\bot_{bool}) &= \top_{bool} \wedge \mathit{ff} \wedge tt \equiv \bot_{bool}. \end{aligned}$$

In this example, as r gets larger, $i(k)(r)$ also gets larger. This property is true in general.

Lemma 2.80 (i Monotone in Second Argument) *If $r \sqsubset t$ and $k? \sqsubset t \to t'$ and*

$$r \leq r'$$

then

$$i(k?)(r) \preceq i(k?)(r').$$

Proof: If $k? = \mathrm{ns}$, then our result follows directly from the definitions of i and $\preceq$. Otherwise $k?$ has the form $k_1 \to k_1' \wedge \ldots \wedge k_n \to k_n'$. As in the definition of i, let

$$s = \{k_j' \mid j \text{ between 1 and } n \text{ and } r \leq k_j\}$$

and

$$s' = \{k_j' \mid j \text{ between 1 and } n \text{ and } r' \leq k_j\}.$$

Since $r \leq r'$ and $\leq$ is transitive, we have $s \supset s'$. Since $k \sqsubset t \to t'$, all of the k_j''s must refine t', so $s \sqsubset t'$. Thus we can use Fact 2.76 ($\triangle$ Elim Sub) on page 107 to get

$$\triangle s \preceq \triangle s'.$$

According to the definition of i, this is our conclusion. □

It is also true that as $k?$ gets larger, $i(k?)(r)$ gets larger, but the proof is somewhat more involved:

Lemma 2.81 (i Monotone in First Argument) *If $r \sqsubset t$ and $k? \sqsubset t \to t'$ and*

$$k? \preceq p?$$

then

$$i(k?)(r) \preceq i(p?)(r).$$

Proof: By induction on the derivation of $k? \preceq p?$.

Case: $p? = \mathbf{ns}$ Then $i(p?)(r) = \mathbf{ns}$, and our result follows from the definition of $\preceq$.

Case: $k? = \mathbf{ns}$ Then $p? = \mathbf{ns}$, so the previous case holds.

Case: SELF-SUB Trivial.

Case: AND-ELIM-R-SUB Since $k? \sqsubset t \to t'$ we know that $k?$ has the form $k_1 \to k_1' \wedge \ldots \wedge k_q \to k_q'$. The shape of AND-ELIM-R-SUB tells us there is some n less than q such that

$$p? = k_1 \to k_1' \wedge \ldots \wedge k_n \to k_n'.$$

Let

$$s = \{k_j' \mid j \text{ between 1 and } q \text{ and } r \leq k_j\}$$

and

$$s' = \{k_j' \mid j \text{ between 1 and } n \text{ and } r \leq k_j\}.$$

Since $n < q$ we have $s \supset s'$. Thus Fact 2.76 ($\triangle$ Elim Sub) on page 107 gives

$$\triangle s \preceq \triangle s',$$

and by definition of i, this is our conclusion.

Case: AND-ELIM-L-SUB Similar.

Case: AND-INTRO-SUB Then $p?$ has the form $p_1 \wedge p_2$, where the premises of AND-INTRO-SUB are

$$k? \leq p_1$$

and

$$k? \leq p_2.$$

Applying our induction hypothesis to these gives

$$i(k?)(r) \preceq i(p_1)(r)$$

and

$$i(k?)(r) \preceq i(p_2)(r).$$

Since $p? = p_1 \wedge p_2$, the definition of i tells us that $i(p?)(r) = i(p_1)(r) \mathbin{\triangle} i(p_2)(r)$. Thus Fact 2.77 ($\triangle$ Intro Sub) on page 107 gives $i(k?)(r) \preceq i(p?)(r)$, which is our conclusion.

Case: TRANS-SUB Then the premises of TRANS-SUB must be

$$k? \leq r'$$

and

$$r' \leq p?.$$

Applying the induction hypotheses to these gives

$$i(k?)(r) \preceq i(r')(r)$$

and

$$i(r')(r) \preceq i(p?)(r).$$

Using Fact 2.78 (Transitivity of $\preceq$) on page 108 on these gives our conclusion.

Case: ARROW-SUB Then $k?$ has the form $k_1 \to k_1'$ and $p?$ has the form $p_1 \to p_1'$ and the premises of ARROW-SUB are

$$p_1 \leq k_1$$

and

$$k_1' \leq p_1'.$$

If $i(p?)(r) = \mathrm{ns}$ then our conclusion follows immediately, so instead suppose that $i(p?)(r)$ is a refinement type. From the definition of i we must have $r \leq p_1$ and $i(p?)(r) = p_1'$. TRANS-SUB and $p_1 \leq k_1$ give $r \leq k_1$, and the definition of i gives $i(k?)(r) = k_1'$. Thus $k_1' \leq p_1'$ is our conclusion.

Case: ARROW-AND-ELIM-SUB Then $k?$ must have the form $k_1 \to k_1' \wedge k_1 \to k_2'$ and $p?$ must have the form $k_1 \to (k_1' \wedge k_2')$. If $i(p?)(r) = \mathrm{ns}$ then the definition of $\preceq$ gives our conclusion immediately. Otherwise the definition of i gives $r \leq k_1$ and

$$i(k?)(r) = k_1' \wedge k_2'$$

and

$$i(p?)(r) = k_1' \wedge k_2'.$$

Before we use SELF-SUB to get our result we must find a u' such that $k_1' \wedge k_2' \sqsubset u'$. The premise of ARROW-AND-ELIM-SUB is

$$k_1 \to (k_1' \wedge k_2') \sqsubset t.$$

This can only be derived by using ARROW-REF, so t must have the form $u \to u'$ and the premises of ARROW-REF must be

$$k_1 \sqsubset u$$

and

$$k_1' \wedge k_2' \sqsubset u'.$$

The latter and SELF-SUB give our conclusion.

Case: RCON-SUB, RCON-AND-ELIM-SUB, TUPLE-SUB, TUPLE-AND-ELIM-SUB

None of these cases can happen because we assume that $k?$ and $p?$ refine a functional ML type. □

This has a simple corollary:

Corollary 2.82 (Bound on Argument to i Gives Bound on i)
If $k \leq r \to p$ then $i(k)(r) \leq p$.

Proof: The definition of i gives $i(r \to p)(r) = p$. By Lemma 2.81 (i Monotone in First Argument) on page 109, $i(k)(r) \leq i(r \to p)(r)$, and rewriting $i(r \to p)(r)$ to p gives our result. □

We will call $r_1, \ldots, r_n$ the *components* of the refinement type $r_1 \wedge \ldots \wedge r_n$.

From the definition of i, it is clear that if $r \to r'$ is one of the components of k, then $i(k)(r) \leq r'$. The converse of this is false; for example, if $k = tt \to ff \wedge tt \to tt$ and $r = tt$, then $i(k)(r) = ff \wedge tt$ but $tt \to (ff \wedge tt)$ is not one of the components of k. However, we do have $k \leq tt \to (ff \wedge tt)$, and this sort of assertion is true in general.

Lemma 2.83 (i Gives an Upper Bound) *If $k? \sqsubset t \to t'$ and $r \sqsubset t_1$ and*

$$i(k?)(r) \preceq r'$$

then

$$k? \leq r \to r'.$$

Proof: We know that $k? \neq \mathrm{ns}$, because if $k?$ were ns then $i(k?)(r)$ would be ns and our hypothesis would be false.

Since $k? \sqsubset t \to t'$, we know that $k?$ has the form $k_1 \to k_1' \wedge \ldots \wedge k_n \to k_n'$. Let

$$S = \{j \text{ between 1 and } n \mid r \leq k_j\}.$$

We know that $i(k?)(r)$ is not ns because $\mathrm{ns} \preceq r'$ cannot be true, so S is not empty. Let

$$k' = \triangle\{k_j \to k_j' \mid j \in S\}.$$

Since all components of k' appear in $k?$, we have

$$k? \leq k'.$$

Let

$$k'' = \triangle\{r \to k'_j \mid j \in S\}.$$

Since each component of k' is a subtype of the corresponding component of k'', we have

$$k' \leq k''.$$

Let

$$k''' = r \to \triangle\{k'_j \mid j \in S\}.$$

Repeated use of ARROW-AND-ELIM-SUB gives

$$k'' \leq k'''.$$

Since $\triangle\{k'_j \mid j \in S\} = i(k?)(r) \leq r'$ is a hypothesis of this lemma, RCON-SUB gives

$$k''' \leq r \to r'.$$

Repeated use of TRANS-SUB gives $k? \leq r \to r'$, which is our conclusion. □

If we have two functions f_1 and f_2 with common domain and codomain, and we can compare elements in the codomain, then we can naturally compare f_1 and f_2 pointwise. That is, f_1 is greater than f_2 if for all x in their common domain, $f_1(x)$ is greater than $f_2(x)$. It turns out that this ordering when used on $i(k)$ is the same as the subtype ordering on k.

Lemma 2.84 (Ordering on i) *If $k?$ and $p?$ refine $t \to t'$ and for all r refining t we have*

$$i(k?)(r) \preceq i(p?)(r)$$

then

$$k? \preceq p?.$$

Proof: If $k? = \mathrm{ns}$, then for all r refining t we have $i(k?)(r) = \mathrm{ns}$, so $i(p?)(r) = \mathrm{ns}$. This can only be the case if $p? = \mathrm{ns}$, so by definition of $\preceq$ we have $k? \preceq p?$, which is our conclusion.

If $p? = \mathrm{ns}$ then we immediately get $k? \preceq p?$ by definition of $\preceq$.

Otherwise, since $p? \sqsubset t \to t'$, we know that $p?$ has the form $p_1 \to p'_1 \wedge \ldots \wedge p_m \to p'_m$. By the definition of i, for all j between 1 and m we have

$$i(p?)(p_j) \leq p'_j.$$

By our hypothesis

$$i(k?)(p_j) \leq i(p?)(p_j).$$

TRANS-SUB gives

$$i(k?)(p_j) \leq p'_j.$$

Lemma 2.83 (i Gives an Upper Bound) on page 111 gives

$$k? \leq p_j \to p'_j.$$

Since this is true for all j, we can repeatedly use AND-INTRO-SUB to get

$$k? \leq p?,$$

which is our conclusion. □

Now we can show that i preserves all the information about its first argument. This is our main result about i; the remainder of the argument after the following lemma is little more than repackaging i to get our interpretation function I, and translating the following lemma to a statement about I.

Lemma 2.85 (i Preserves Information) *Suppose $r?$ and $r?'$ both refine $t \to t'$. Then*

$$\textit{for all } k \textit{ and } k' \textit{ refining } t \textit{ we have } k \equiv k' \textit{ implies } i(r?)(k) \approx i(r?')(k') \tag{2.46}$$

if and only if

$$r? \equiv r?'. \tag{2.47}$$

Proof of (2.46) implies (2.47): From (2.46) we get

$$\text{for all } k \text{ and } k' \text{ refining } t \text{ we have } k \equiv k' \text{ implies } i(r?)(k) \preceq i(r?')(k')$$

Since $k \sqsubset t$, we can choose $k' = k$ and we have $k \equiv k$. Thus

$$\text{for all } k \text{ refining } t \text{ we have } i(r?)(k) \preceq i(r?')(k)$$

and Lemma 2.84 (Ordering on i) on page 112 gives

$$r? \leq r?'.$$

A symmetric argument gives

$$r?' \leq r?,$$

and together these imply our conclusion.

Proof of (2.47) implies (2.46): From (2.47) we get

$$r? \leq r?',$$

and we can then use Lemma 2.81 (i Monotone in First Argument) on page 109 to get

$$i(r?)(k) \preceq i(r?')(k).$$

The premise of the implication in (2.46) gives

$$k \leq k',$$

and Lemma 2.80 (i Monotone in Second Argument) on page 108 gives

$$i(r?')(k) \preceq i(r?')(k').$$

Fact 2.78 (Transitivity of $\preceq$) on page 108 gives

$$i(r?)(k) \preceq i(r?')(k').$$

A symmetric argument gives

$$i(r?')(k') \preceq i(r?)(k),$$

and together these are our conclusion. □

Now we will repackage i as a function mapping equivalence classes to equivalence classes. First we must define some notation for manipulating equivalence classes:

Definition 2.86 *If $r?$ is a generalized refinement type, then the equivalence class $C(r?)$ containing $r?$ is the set $\{r?' \mid r?' \approx r?\}$.*

Definition 2.87 *If t is an ML type, then EC(t) is the set of equivalence classes of generalized refinements of t, or in symbols, $\{C(r?) \mid r? \sqsubset t\}$.*

We shall use c as a metavariable standing for the equivalence class of a refinement type, and $c?$ as a metavariable standing for the equivalence class of a generalized refinement type.

Now we have the machinery to define the interpretation of refinement types as a mapping from equivalence classes to equivalence classes:

Definition 2.88 *If $c?' \in$ EC(t') and $c \in$ EC(t) and $r? \sqsubset t \to t'$, then we write*

$$c?' = I(r?)(c)$$

if there is a $k \in c$ such that $c?' = C(i(r?)(k))$.

By Lemma 2.80 (i Monotone in Second Argument) on page 108, we know that $i(r?)$ is a function that maps equivalent refinement types to equivalent refinement types. Thus $I(r?)$ is a function. We can also show that I maps equivalent refinement types to equal functions:

Lemma 2.89 (I Preserves Equivalence) *Suppose $k?$ and $p?$ refine $t \to t'$. Then*

$$I(k?) = I(p?)$$

if and only if

$$k? \approx p?.$$

Proof: By definition of equality for functions we have $I(k?) = I(p?)$ if and only if for all c in $EC(t)$ we have $I(k?)(c) = I(p?)(c)$. By definition of I, this is true if and only if whenever $r \equiv r'$ we have $i(k?)(r) \approx i(p?)(r')$. By Lemma 2.85 ($i$ Preserves Information) on page 113, this is equivalent to $k? \equiv p?$. □

Theorem 2.90 (Finite Refinements) *For each ML type u we have EC(u) is finite.*

Proof: By induction on u.

Case: u=*uc* By Assumption 2.8 (Finite Predefined Refinements) on page 31, uc has only finitely many refinements, so it can have only finitely many equivalence classes.

Case: $u = t_1 * \ldots * t_n$ Any refinement of u must have the form

$$(r_1^1 * \ldots * r_n^1) \wedge \ldots \wedge (r_1^m * \ldots * r_n^m).$$

By Fact 2.23 (Tuplesimp Sound) on page 41, this is equivalent to a refinement type of the form $k_1 * \ldots * k_n$. By TUPLE-SUB, two refinements of u of this form are equivalent if and only if they are equivalent componentwise. Since our induction hypothesis tells us that there are only finitely many equivalence classes for each component, there are only finitely many equivalence classes of tuples without a toplevel $\wedge$. Since each refinement of u is equivalent to one without a toplevel $\wedge$, there are only finitely many equivalence classes of refinements of u.

Case: $t = u \to u'$ By our induction hypothesis, $EC(u)$ and $EC(u')$ are both finite. By Lemma 2.89 (I Preserves Equivalence) on page 114, for all r refining t we have $C(r)$ is uniquely determined by $I(r)$. Since $I(r)$ maps elements of $EC(u)$ to elements of $EC(u')$, there are only finitely many distinct values for $I(r)$, and therefore only finitely many values of $C(r)$ and only finitely many values in $EC(t)$. □

Finite Refinements straightforwardly gives us principal refinement types. Later on we prove that there is an algorithm that computes principal refinement types; this proof can also be interpreted as a proof that principal types exist, but it has the disadvantage of being much more complex than the simple proof we give here.

Corollary 2.91 (Principal Refinement Types) *If*

$$\text{VR} \vdash e : r$$

then there is a k such that

$$\text{VR} \vdash e : k$$

and for all p we have

$$\text{VR} \vdash e : p \textit{ implies } k \leq p.$$

We prove this by choosing k to be the intersection of all refinement types such that VR $\vdash e : k$, with suitable perturbations to ensure that this intersection is finite.

Proof: By Theorem 2.54 (Inferred Types Refine) on page 68, there is a t such that $r \sqsubset t$ and

$$\text{rtom(VR)} \vdash e :: t. \tag{2.48}$$

Let

$$sc = \{c \in EC(t) \mid \text{for some } r \text{ in } c \text{ we have VR} \vdash e : r\}.$$

and for c in sc let k_c be an arbitrary but fixed element of c. By Theorem 2.90 (Finite Refinements) on page 115, $EC(t)$ is finite. Thus sc is finite and we can choose

$$k = \wedge\{k_c \mid c \in sc\}$$

without creating an infinite syntactic object. Now we have to prove that k has the two properties required by our conclusions.

Proof of VR $\vdash e : k$**:** By construction of sc, for each k_c there is a k'_c such that $k_c \equiv k'_c$ and

$$\text{VR} \vdash e : k'_c.$$

WEAKEN-TYPE immediately gives VR $\vdash e : k_c$, and repeated use of AND-INTRO-TYPE gives

$$\text{VR} \vdash e : k.$$

Proof of VR $\vdash e : p$ **implies** $k \leq p$**:** Suppose VR $\vdash e : p$. By Theorem 2.54 (Inferred Types Refine) on page 68, there is a u such that $p \sqsubset u$ and rtom(VR) $\vdash e :: u$. By Lemma 2.4 (Unique Inferred ML Types) on page 27, we must have $u = t$. Thus, by construction of sc, p must be in some equivalence class c in sc. Since c is an equivalence class, $k_c \equiv p$. By repeated use of AND-ELIM-L-SUB and AND-ELIM-R-SUB, we have

$$k \leq p.$$

Since this argument is valid whenever VR $\vdash e : p$, we have

$$\text{VR} \vdash e : p \text{ implies } k \leq p,$$

which is our second conclusion. □

2.10 Decidability

This section will describe an algorithm for finding the principal refinement type of an expression. This requires being able to list one representative of each equivalence class of refinements of an ML type and being able to decide whether one refinement type is a subtype of another. These last two algorithms are mutually recursive, so we will describe them together in Subsection 2.10.1. Then we will give an algorithm for finding the principal split of a refinement type in Subsection 2.10.2. A notion of least upper bound for refinement types is defined in Subsection 2.10.3, then we use all of these in an algorithm for deciding refinement types in Subsection 2.10.4.

2.10.1 Deciding Subtyping and Enumerating Refinements

Now we will describe procedures for determining whether one refinement type is a subtype of another and for enumerating the refinements of an ML type. The strategies for doing both of these are very straightforward except when we are dealing with functional ML types.

To determine whether $r \leq k$ when both r and k refine a functional ML type $t_1 \to t_2$, we check whether, for all p refining t_1, we have $i(r)(p) \preceq i(k)(p)$. If this condition is true for all p, then $r \leq k$.

To enumerate all refinements of a functional ML type $t_1 \to t_2$, we enumerate all possible monotone functions $i(r)$ mapping refinements of t_1 to generalized refinements of t_2, and convert each function to a refinement type. Converting these functions to refinement types is the job of the `fntoref` procedure described below; this procedure is the inverse of i, since when we only specify the first argument of i, it maps a refinement type to a monotone function from refinement types to generalized refinement types.

We will describe the algorithms for computing this in stages. In this introduction we will briefly list the functions and give an intuitive idea of what they should do. Then in Subsubsection 2.10.1.1 we will give a formal specification of the functions along with pseudocode implementing them. Finally, in Subsubsection 2.10.1.2 we will prove each function satisfies its specification.

The functions involved are:

$\triangle$`fn` s Computes the intersection of refinement types in s.

`botfn` t Computes the least refinement of t.

`allrefs` t Returns a set containing one representative from each equivalence class of refinements of t.

`subtypep` $r?$ $k?$ t Determines whether $r? \preceq k?$, assuming both $r?$ and $k?$ refine t.

`ifn` $r?$ p t Computes $i(r?)(p)$, assuming $p \sqsubset t$ and for some u we have $r? \sqsubset t \to u$.

`fntoref` f t If f is a monotone function from refinements of t to generalized refinements of some u, and r refines t, then $f(r) \approx i($`fntoref` $f)(r)$.

2.10.1.1 Specifications and Definitions

We will describe the algorithm by using a mixture of SML and mathematical notation. In this notation, we use braces ($\{\ \}$) to denote mathematical sets, not SML records. We will also freely use ellipses ($\ldots$) and set comprehensions ($\{\ |\ \}$) when the meaning is obvious and obviously computable. We will assume an infix operator $\times$ takes the cross product of several sets of refinement types and combines them into tuples; for example,

$$\{tt, \mathit{ff}\} \times \{\top_{bool}\} \times \{runit\} = \{tt * \top_{bool} * runit, \mathit{ff} * \top_{bool} * runit\}.$$

$\triangle$`fn` First we have a trivial utility procedure to compute $\triangle$ and give an example of our notation for algorithms. If s is a finite set of refinement types, the function call $\triangle$`fn` s must return $\triangle s$. For example,

$$\triangle\texttt{fn}\ \{tt, \mathit{ff}\} = tt \wedge \mathit{ff}.$$

We define $\triangle$`fn` as follows:

$$\begin{array}{l}\texttt{fun}\ \triangle\texttt{fn}\ \{\}\ \texttt{=}\ \mathbf{ns} \\ \quad \texttt{|}\ \triangle\texttt{fn}\ \{r\}\ \texttt{=}\ r \\ \quad \texttt{|}\ \triangle\texttt{fn}\ (\{r, k\} \cup s)\ \texttt{=}\ r \wedge (\triangle\texttt{fn}\ (\{k\} \cup s))\end{array}$$

The last case may be a little confusing because we are using SML's destructuring notation to destructure a mathematical object. It means "whenever the argument to $\triangle$`fn` has the form $\{r, k\} \cup s$, the result is $r \wedge (\triangle\texttt{fn}\ (\{k\} \cup s))$". This notation is vague about which elements of s we choose to name r and k; this vagueness (and similar vagueness in algorithms that follow) makes no important difference, and we shall ignore it.

The above function is the only one that does not participate in the mutual recursion to come.

`botfn` The function call `botfn` t returns the least refinement of t. That is, if $r \sqsubset t$, then $(\texttt{botfn}\ t) \leq r$. The code for `botfn` is:

$$\texttt{fun botfn}\ t\ \texttt{=}\ \wedge(\texttt{allrefs}\ t)$$

Note that there may be values of ML type t with the refinement type `botfn` t; for example, if have a datatype *d* and with no declared refinements, there will be only one refinement of *d*; call it $\top_d$. Then `botfn` $d = \top_d$, which will be inhabited by all values with ML type d.

`allrefs` The function call `allrefs` t returns a list of one representative of each equivalence class of refinements of t the refinements of t. If $r \sqsubset t$, then there must be a $k \in$ `allrefs` t such that $r \equiv k$. Also, for all t we must have `allrefs` t is finite.

For example,

$$\texttt{allrefs}\ \mathit{bool} = \{tt, \mathit{ff}, \top_{\mathit{bool}}, \bot_{\mathit{bool}}\}.$$

It would be consistent with the specification for `allrefs` bool to return $\{tt \wedge tt, \mathit{ff} \wedge \top_{\mathit{bool}}, \top_{\mathit{bool}}, tt \wedge \mathit{ff}\}$. The code for `allrefs` follows. As in Standard ML, when we have multiple mutually recursive procedures, we introduce all but the first with the `and` keyword instead of the `fun` keyword.

```
and allrefs (t1 * ... * tn) = allrefs t1 × ... × allrefs tn
  | allrefs (t → u) =
        {fntoref f t |
          f is a function from allrefs t to (allrefs u) ∪ {ns}
          and f(botfn t) ≠ ns
          and for all r and k in allrefs t we have
                subtypep r k t implies subtypep (f r) (f k) u}
  | allrefs tc = {rc | rc ⊑(def) tc}
```

`subtypep` The function call `subtypep` $r?$ $k?$ t determines whether $r? \preceq k?$, assuming that both $r?$ and $k?$ refine t. For example,

$$\texttt{subtypep}\ tt\ ff\ bool = \texttt{true}.$$

The code for `subtype` follows; note that it uses the `tuplesimp` and `rconsimp` functions, which are defined on pages 41 and 42, respectively.

```
and subtypep _ ns _ = true
  | subtypep ns k _ = false
  | subtypep r k (t1 * ... * th) =
        let val r1 * ... * rh = tuplesimp r
            val k1 * ... * kh = tuplesimp k
        in
            for j in 1...h we have subtypep rj kj tj
        end
  | subtypep r k (t1 → t2) =
        for all p ∈ allrefs t1 we have subtypep (ifn r p t1) (ifn k p t1) t2
  | subtypep r k tc =
        let val rc = rconsimp r
            val kc = rconsimp k
        in
            rc ≤(def) kc
        end
```

We could make the $t_1 \rightarrow t_2$ case more efficient by replacing it with

```
  | subtypep r k (t1 → t2) =
        let (k1 → k1' ∧ ... ∧ kn → kn') = k
        in
            for all j in 1...n we have subtypep (ifn r kn t1) kn' t2
        end
```

but this would be slightly more difficult to prove, and it does not work for the representation of refinement types used in the serious exploration of efficiency in Chapter 7. Thus we will stay with the simpler but less efficient version.

`ifn` The function call `ifn` $r?\ p\ t$ computes $i(r?)(p)$, assuming that for some u we have $r? \sqsubset t \to u$ and $p \sqsubset t$. For example,

$$\texttt{ifn}\ (tt \to tt \wedge \mathit{ff} \to \mathit{ff})\ \perp_{bool}\ \mathit{bool} = tt \wedge \mathit{ff}$$

and

$$\texttt{ifn}\ (tt \to tt)\ \mathit{ff}\ \ \mathit{bool} = \mathrm{ns}.$$

The code for `ifn` is:

```
and ifn r? p t =
   if r? = ns then ns
   else
        let val r1 → r1' ∧ ... ∧ rn → rn' = r?
        in △fn {rh' | h ∈ 1...n and subtypep p rh t}
        end
```

`fntoref` The function `fntoref` is an inverse of sorts to `ifn`. If f maps refinements of t to generalized refinements of some other ML type, and f is monotone, and $f(\texttt{botfn}\ t)$ is not ns, then `fntoref` $f\ t$ is a refinement type and for all k refining t we have

$$f(k) \approx i(\texttt{fntoref}\ f\ t, k).$$

We need to require f to be monotone because i is monotone in its second argument, so the equivalence just displayed could not possibly be true if f is not monotone. We need $f(\texttt{botfn}\ t)$ to be something other than ns to ensure `fntoref` $f\ t$ is always a refinement type. The code for `fntoref` is:

```
and fntoref f t =
        △fn {r → f(r) | r ∈ allrefs t and f(r) ≠ ns}
```

2.10.1.2 Soundness

To prove these algorithms sound, we need to prove they always terminate and they fit their specifications. First we will show partial correctness, then we will give an informal proof of termination.

Theorem 2.92 (Subtype Decidability) *All of the functions discussed in Subsubsection 2.10.1.1 fulfill their specification when they terminate.*

Proof: By induction on the evaluation of the function.

$\triangle$`fn` meets its specification Trivial.

`botfn` meets its specification We need to show that if $r \sqsubset t$, then `botfn` $t \leq r$. By induction hypothesis, we can assume that `allrefs` is sound; thus there is a k in `allrefs` t such that $r \equiv k$. By repeated use of AND-ELIM-L-SUB and AND-ELIM-R-SUB we have $\wedge$(`allrefs` t) $\leq k$. Then TRANS-SUB gives $\wedge$(`allrefs` t) $\leq r$, which is our conclusion.

`allrefs` meets its specification We need to show that for all t we have `allrefs` t is finite, and that if $r \sqsubset t$, there is a $k \in$ `allrefs` t such that $r \equiv k$.

Case: `allrefs` $(t_1 * \ldots * t_n)$ The code for this case is

$$\texttt{fun allrefs } (t_1 * \ldots * t_n) \texttt{ = allrefs } t_1 \times \ldots \times \texttt{allrefs } t_n$$

Suppose $r \sqsubset t_1 * \ldots * t_n$. Then `tuplesimp` r must be defined and have the form $r_1 * \ldots * r_n$, and soundness of `tuplesimp` gives `tuplesimp` $r \equiv r_1 * \ldots * r_n$. By induction hypothesis, for h in $1 \ldots n$ there is a p_h in `allrefs` t_h such that $r_h \equiv p_h$. Then TUPLE-SUB gives

$$r_1 * \ldots * r_n \equiv p_1 * \ldots * p_n$$

and TRANS-SUB gives

$$r \equiv p_1 * \ldots * p_n.$$

The definition of $\times$ then gives

$$p_1 * \ldots * p_n \in \texttt{allrefs } t_1 \times \ldots \times \texttt{allrefs } t_n.$$

Thus $p_1 * \ldots * p_n$ is the member of `allrefs` t that we seek. Since the cross product of a finite number of finite sets is finite, `allrefs` t is a finite set, so we are done.

Case: `allrefs` $t \to u$ The code for this case is

```
| allrefs (t → u) =
      {fntoref f  t |
        f  is a function from  allrefs t  to  (allrefs u) ∪ {ns}
        and  f(botfn t) ≠ ns
        and  for all r and k in allrefs t we have
                subtypep r  k  t implies subtypep (f  r)  (f  k)  u}
```

Suppose $r \sqsubset t \to u$. Then r has the form $r_1 \to r_1' \wedge \ldots \wedge r_n \to r_n'$. Define

$$f = \lambda k. \begin{cases} \mathbf{ns} & \text{if } i(r,k) = \mathbf{ns} \\ \text{any } p \text{ in } \texttt{allrefs } u \text{ such that } p \equiv i(r,k) & \text{otherwise.} \end{cases}$$

We will show that `fntoref` f t is in `allrefs` $t \to u$, and that $r \equiv$ `fntoref` f t.

By definition of i, we know that $f($`botfn` $t)$ is $r'_1 \wedge \ldots \wedge r'_n$, which is certainly not ns. Since $i(r, k)$ is monotone in k we know that f is monotone. Thus the code for this case of `allref` tells us

$$\texttt{fntoref}\ f\ t \text{ is in } \texttt{allrefs}\ (t \to u).$$

Let $r' =$ `fntoref` f t. Since `fntoref` is sound, we know that

$$\text{for all } k \text{ refining } t \text{ we have } f(k) \approx i(r', k).$$

By definition of f we have

$$\text{for all } k \text{ refining } t \text{ we have } f(k) \approx i(r, k).$$

By transitivity of $\approx$, these imply

$$\text{for all } k \text{ refining } t \text{ we have } i(r', k) \approx i(r, k).$$

By Lemma 2.85 (i Preserves Information) on page 113, this implies $r' \equiv r$.

We know that `allrefs` $t \to u$ is finite because by induction `allrefs` t and `allrefs` u are finite, and there are finitely many functions from a finite set to a finite set.

Case: `allrefs` tc The code for this case is

$$\texttt{| allrefs}\ tc\ \texttt{=}\ \{rc \mid rc \stackrel{\text{def}}{\sqsubset} tc\}$$

By Assumption 2.8 (Finite Predefined Refinements) on page 31, `allrefs` tc is finite.

Suppose $r \sqsubset tc$. Then r must have the form $rc_1 \wedge \ldots \wedge rc_n$, where for all i, $rc_i \sqsubset tc$. Then Lemma 2.24 (Refinement Constructor Intersection) on page 41 gives

$$r \equiv rc_1 \stackrel{\text{def}}{\wedge} \ldots \stackrel{\text{def}}{\wedge} rc_n.$$

By Theorem 2.21 (Subtypes Refine) on page 36, $rc_1 \stackrel{\text{def}}{\wedge} \ldots \stackrel{\text{def}}{\wedge} rc_n \sqsubset tc$; this can only be inferred by using RCON-REF with the premise $rc_1 \stackrel{\text{def}}{\wedge} \ldots \stackrel{\text{def}}{\wedge} rc_n \stackrel{\text{def}}{\sqsubset} tc$. Thus $rc_1 \stackrel{\text{def}}{\wedge} \ldots \stackrel{\text{def}}{\wedge} rc_n \in$ `allrefs` tc, which is what we wanted to show.

`subtypep` **meets its specification** We need to show that if both $r?$ and $k?$ refine t, then `subtype` $r?$ $k?$ t returns true if and only if $r? \preceq k?$.

Case: `subtypep` $r?$ `ns _` The code for this case is

```
and subtypep _ ns _ = true
```

The definition of $\preceq$ gives $r? \preceq \mathrm{ns}$, which is what we wanted to show.

Case: `subtypep` ns k? `_` The code for this case is

```
| subtypep ns _ _ = false
```

The definition of $\preceq$ tells us that $\mathrm{ns} \preceq k?$ is false, which is what we wanted to show.

Case: `subtypep` r k $(t_1 * \ldots * t_n)$ The code for this case is

```
| subtypep r k (t1 * ... * th) =
      let val r1 * ... * rh = tuplesimp r
          val k1 * ... * kh = tuplesimp k
      in
          for j in 1...h we have subtypep rj kj tj
      end
```

Since r and k both refine $t_1 * \ldots * t_n$, `tuplesimp` r and `tuplesimp` k are defined and

`tuplesimp` r has the form $r_1 * \ldots * r_h$

and

`tuplesimp` k has the form $k_1 * \ldots * k_h$.

By soundness of `tuplesimp`,

$$r \equiv r_1 * \ldots * r_h$$

and

$$k \equiv k_1 * \ldots * k_h.$$

Suppose

$$r \leq k. \tag{2.49}$$

By TRANS-SUB, this is equivalent to

$$r_1 * \ldots * r_h \leq k_1 * \ldots * k_h.$$

By TUPLE-SUB and Corollary 2.27 (TUPLE-SUB Inversion) on page 45, this is equivalent to

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_j \leq k_j.$$

By induction hypothesis, `subtypep` is sound, so this is equivalent to

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } \texttt{subtypep } r_j \;\; k_j \;\; t_j\,. \tag{2.50}$$

Summarizing the above argument, (2.49) is equivalent to (2.50). By definition of `subtype`, this is our conclusion.

Case: `subtypep` r k $(t_1 \to t_2)$ The code for this case is

```
| subtypep r k (t1 → t2) =
      for all p ∈ allrefs t1 we have subtypep (ifn r p t1) (ifn k p t1) t2
```

Suppose

$$r \leq k \tag{2.51}$$

By Lemma 2.84 (Ordering on i) on page 112 and Lemma 2.81 (i Monotone in First Argument) on page 109, this is equivalent to

$$\text{for all } p' \sqsubset t_1 \text{ we have } i(r)(p') \preceq i(k)(p'). \tag{2.52}$$

Since we can assume by induction that recursive calls to `allrefs` are sound, for all $p' \sqsubset t_1$ there is a p in `allrefs` t_1 such that $p \equiv p'$. This and Lemma 2.81 (i Monotone in First Argument) on page 109 give

$$\begin{gathered}\text{for all } p' \sqsubset t_1 \text{ there is a } p \text{ in } \texttt{allrefs}\ t_1 \text{ such that} \\ i(r)(p) \approx i(r)(p') \text{ and } i(k)(p) \approx i(k)(p').\end{gathered}$$

We can use this with Fact 2.78 (Transitivity of $\preceq$) on page 108 on (2.52) to get

$$\text{for all } p \text{ in } \texttt{allrefs}\ t_1 \text{ we have } i(r)(p) \preceq i(k)(p).$$

By induction, we can assume that recursive calls to `subtypep` and `ifn` are sound, so this is equivalent to

$$\text{for all } p \text{ in } \texttt{allrefs}\ t_1 \text{ we have } \texttt{subtypep}\ (\texttt{ifn}\ r\ p\ t_1)\ (\texttt{ifn}\ k\ p\ t_1)\ t_2 \tag{2.53}$$

Summarizing the argument so far, (2.51) is equivalent to (2.53). This is our conclusion.

Case: `subtypep` r k tc

The code for this case is

```
| subtypep r k tc =
      let val rc = rconsimp r
          val kc = rconsimp k
      in
          rc ≤(def) kc
      end
```

By assumption, r and k refine tc; therefore both calls to `rconsimp` are valid. By Fact 2.23 (Tuplesimp Sound) on page 41, $r \equiv rc$ and $k \equiv kc$.

Suppose $r \leq k$. By TRANS-SUB, this is equivalent to $rc \leq kc$; by RCON-SUB and Fact 2.29 (RCON-SUB Inversion) on page 45, this is equivalent to $rc \overset{\text{def}}{\leq} kc$. Summarizing the argument so far, $r \leq k$ if and only if $rc \overset{\text{def}}{\leq} kc$. By definition of `subtypep`, this is our conclusion.

`ifn` meets its specification The code for `ifn` is

```
and ifn r p t =
        let val r1 → r1' ∧ ... ∧ rn → rn' = r
        in △fn {rh' | h ∈ 1...n and subtypep p rh t}
        end
```

and we need to show that if $r \sqsubset t \to u$ and $p \sqsubset t$ then `ifn` $r\ p\ t = i(r,p)$. This is obviously correct, since we can assume by induction that the recursive calls to `subtypep` are sound.

`fntoref` meets its specification The code for `fntoref` is

```
and fntoref f t =
        △fn {r → f(r) | r ∈ allrefs t and f(r) ≠ ns}
```

and we need to show that if there is a u such that

$$f \text{ maps refinements of } t \text{ to generalized refinements of } u$$

and

$$f \text{ is monotone}$$

and

$$f(\texttt{botfn}\ t) \text{ is not ns},$$

then for all $k \sqsubset t$ we have

$$f(k) \approx i(\texttt{fntoref}\ f\ t)(k).$$

Since $f(\texttt{botfn}\ t)$ is not ns, we know that $(\texttt{botfn}\ t) \to f(\texttt{botfn}\ t)$ is one of the components in `fntoref` $f\ t$. Thus `fntoref` $f\ t$ is not ns, and we can let $r =$ `fntoref` $f\ t$.

Suppose $k \sqsubset t$. By definition of i,

$$i(r)(k) = \triangle\{f(p) \mid k \leq p\}.$$

SELF-SUB gives $k \leq k$, so $f(k)$ is in $\{f(p) \mid k \leq p\}$ and Fact 2.77 ($\triangle$ Intro Sub) on page 107 gives

$$f(k) \leq \triangle\{f(p) \mid k \leq p\}.$$

Since f is monotone, $k \leq p$ implies $f(k) \leq f(p)$. Thus all elements of $\{f(p) \mid k \leq p\}$ are greater than $f(k)$, so Fact 2.76 ($\triangle$ Elim Sub) on page 107 gives

$$\triangle\{f(p) \mid k \leq p\} \leq f(k).$$

Thus $i(r)(k) \approx f(k)$, which is our conclusion. □

Theorem 2.93 (Termination for `subtypep` and `allrefs`) *All algorithms defined in Subsubsection 2.10.1.1 terminate for all inputs.*

The function $\triangle$`fn` terminates because its argument is a finite set. An infinite execution of any other function must involve infinitely many recursive calls to some function, call it f. Examination of the code tells us f must have an argument that is an ML type, and that ML type must get smaller from one call to f to the next. Since all ML types are finite, this is a contradiction. Thus all executions are finite. □

2.10.2 Deciding Splits

In the refinement type inference algorithm we present in Subsection 2.10.4, the SPLIT-TYPE rule is always done as early as possible; each variable is split exactly once when it is added to the variable environment. For example, if the algorithm is considering what might happen if x has type $\top_{bool}$, it will split this into the possibilities $x : tt$ and $x : ff$ when x is added to the environment, and it will not consider splitting x again. The appropriate split to use is the principal split of the type of x, as discussed in Subsubsection 2.6.2.2. This Subsection gives a procedure for computing principal splits.

2.10.2.1 Computing Principal Splits

We will give a constructive proof that principal splits exist which can also be used as an algorithm for computing them. But first we will give an algorithm `anysplit` that returns a useful split of a refinement type if there is one. If there are none, then `anysplit` returns the singleton set containing its argument.

```
fun anysplit r (t1 * ... * th) =
        let val r1 * ... * rh = tuplesimp r
        in
           anysplit r1 × ... × anysplit rh
        end
  | anysplit r tc =
        let val rc = rconsimp r
            in
               if rc has a useful predefined split sc
               then sc
               else {r}
            end
  | anysplit r (t1 → t2) = {r}
```

Soundness of his algorithm follows by induction on the ML type argument.

Assuming `anysplit` works, it is straightforward to find a principal split. Just split all of the fragments so long as there is one with a useful split, and then use ELIM-SPLIT to eliminate as many elements as possible.

Theorem 2.94 (Principal Split Existence) *If $r \sqsubset t$ and $r \asymp s$ then we can construct an s' that is a principal split of r.*

Proof: Let $s_1 = s$. For $i \geq 1$, if there is an element r_i of s_i with a useful split s_i'', then let

$$s_{i+1} = (s_i - \{r_i\}) \cup s_i''.$$

This process has to stop eventually, because by definition of useful the elements added to each s_i are strictly subtypes of the elements we take from s_i, and t has only finitely many refinements. Let n be the last value of i; by construction, we know that

$$\text{for all } k \text{ in } s_n, \text{ all splits of } k \text{ are useless.}$$

By repeated use of TRANS-SPLIT,

$$\text{for all } i \text{ we have } r \asymp s_i.$$

Once we eliminate as many elements as possible from s_n we will have our result. This is straightforward: arbitrarily order the elements of s_n such that

$$\{k_1, \ldots, k_m\} = s_n.$$

Let

$$s' = \{k_j \mid j \text{ in } 1 \ldots m \text{ and whenever } k_h \in s_n \text{ and } k_j \leq k_h \text{ we have } k_j \equiv k_h \text{ and } h \geq j.\}.$$

By construction, ELIM-SPLIT can eliminate no more elements from s'. By repeated use of ELIM-SPLIT, $r \asymp s'$. Since s' is a subset of s,

$$\text{for all } k \text{ in } s', \text{ all splits of } k \text{ are useless.}$$

Thus, by Lemma 2.46 (Fragments of Principal Split have Useless Splits) on page 58, s' is a principal split of r. □

2.10.3 Join

A refinement type for a `case` statement is an upper bound of the refinement types of the reachable branches, and the principal refinement type of the case statement is the least upper bound of the principal refinement types of the reachable branches. Therefore we need to be able to compute least upper bounds of refinement types. For example, assume that we have an ML datatype type $blist$ with only the refinement $\top_{blist}$ and a function `emptyp` with refinement type $\top_{blist} \rightarrow \top_{bool}$ and a value `nil ()` of type $\top_{blist}$. The intuition is that `emptyp` determines whether the list is empty, but the programmer has not declared interest in the distinction between empty $blist$'s and nonempty $blist$'s, so refinement type inference will not notice this. Then the expression `emptyp (nil ())` has the principal type $\top_{bool}$, and computing the principal type of the statement

```
case emptyp (nil ()) of
   true => fn ignored:tunit => true ()
 | false => fn ignored:tunit => false ()
end:bool
```

requires finding the least upper bound of the principal types of the expressions `true ()` and `false ()`, yielding $\top_{bool}$. In general, the least upper bound will not always exist; in that case the `case` statement has no refinement type. For example, assuming `x` has the type $tt \to tt$ and `y` has the type $ff \to ff$, trying to find a type for the statement

```
case emptyp (nil ()) of
   true => fn ignored:tunit => x
 | false => fn ignored:tunit => y
end:bool → bool
```

requires finding a least upper bound for $tt \to tt$ and $ff \to ff$. There is none, and this statement has no refinement type. (The reader may object that we cannot write an expression that has the principal type $tt \to tt$. This is true for the language constructs introduced in this chapter, but it will be false after we introduce the $\triangleleft$ operator in Chapter 6. In any case, it is consistent with the theory to hypothesize such a variable.)

We will call these least upper bounds "joins" rather than "disjunctions" or "unions". Calling them disjunctions would conflict with existing nomenclature used in type theory. Calling them unions would be misleading because if we interpret the refinement types as sets, the interpretation of the join of two refinement types may be a proper superset of the union of their interpretations. For example, the join of the refinement types $\top_{blist} \to tt$ and $\top_{blist} \to ff$ is $\top_{blist} \to \top_{bool}$. The function `emptyp` is in the interpretation of $\top_{blist} \to \top_{bool}$, but it is not in the interpretations of either $\top_{blist} \to tt$ or $\top_{blist} \to ff$, so it is not in the union of their interpretations.

According to John Reynolds [personal communication, 1993], type inference for Forsythe uses a similar notion of least upper bounds for the same purpose.

2.10.3.1 Definition of Join

The least upper bound, if it exists, is the greatest lower bound of all of the upper bounds.

Definition 2.95 *If* $r? \sqsubset t$ *and* $k? \sqsubset t$ *then we define*

$$r? \sqcup k? = \triangle\{p \in \texttt{allrefs}\ t \mid r? \preceq p \text{ and } k? \preceq p\}.$$

We would say "$p \sqsubset t$" instead of "$p \in$ `allrefs` t" except $\triangle$ is meaningless for infinite sets, and if we compare refinements of t by mathematical equality rather than refinement type equivalence, there are infinitely many refinements of t.

We also define a join operator for refinement type constructors:

Definition 2.96 *If* $rc \stackrel{\text{def}}{\sqsubset} tc$ *and* $kc \stackrel{\text{def}}{\sqsubset} tc$ *then we define*

$$rc \stackrel{\text{def}}{\sqcup} kc = \stackrel{\text{def}}{\bigwedge} \{pc \mid rc \stackrel{\text{def}}{\leq} pc \text{ and } kc \stackrel{\text{def}}{\leq} pc\}.$$

If the set is empty, then $rc \stackrel{\text{def}}{\sqcup} kc$ is undefined.

It is easy to see that $r? \sqcup k?$ is an upper bound of $r?$ and $k?$ in the $\preceq$ ordering; we can derive $r? \preceq r? \sqcup k?$ by using the definition of $\sqcup$ and repeated use of Fact 2.77 ($\triangle$ Intro Sub) on page 107.

It is also easy to see that it is a least upper bound; if p is an upper bound of $r?$ and $k?$, it is one of the components in $r? \sqcup k?$, so repeated use of AND-ELIM-R-SUB and AND-ELIM-L-SUB gives $r? \sqcup k? \preceq p$. The definition of $\preceq$ tells us that ns is also an upper bound of $r?$ and $k?$ and that $r? \sqcup k? \preceq \mathrm{ns}$.

We can effectively compute $\sqcup$ because `allrefs` and $\preceq$ are both computable. Unfortunately, the obvious algorithm derived directly from the definition is inefficient because the size of the set returned by `allrefs` is exponential in the size of the argument to `allrefs`. In this section we will present a more efficient algorithm.

If the obvious slow algorithm were used to find $tt \to tt \sqcup tt \to \mathit{ff}$, it would first find `allrefs` $(bool \to bool)$ and then take the intersection of all types in that set that are greater than both $tt \to tt$ and $tt \to \mathit{ff}$. The algorithm presented in this subsection would only need to evaluate `allrefs` $bool$. The algorithm presented here is still exponential though; for instance, it will evaluate `allrefs` $(bool \to bool)$ if asked to find $(tt \to tt) \to tt \sqcup (tt \to \mathit{ff}) \to tt$.

Theorem 2.97 (Join is Decidable) *There is an algorithm* `joinf` *mapping two generalized refinement types and an ML type to a generalized refinement type such that if*

$$r? \sqsubset t$$

and

$$k? \sqsubset t$$

then all of the following are true:

$$r? \preceq p \text{ and } k? \preceq p \text{ implies } \texttt{joinf}\ r?\ k?\ t \preceq p$$

$$r? \preceq \texttt{joinf}\ r?\ k?\ t$$

$$k? \preceq \texttt{joinf}\ r?\ k?\ t$$

computation of `joinf` $r?\ k?\ t$ *terminates.*

Proof: We will present the definition of `joinf` as we prove it correct. The proof is by induction on t. In each case we will omit the proof of $k? \preceq \texttt{joinf}\ r?\ k?\ t$ because it is essentially the same as the proof of $r? \preceq \texttt{joinf}\ r?\ k?\ t$.

Since least upper bounds are unique, our conclusion is equivalent to

$$\texttt{joinf}\ r?\ k?\ t \approx r? \sqcup k?.$$

Case: `joinf ns _` or `joinf _ ns` This case reads

```
fun joinf ns _ _ = ns
  | joinf _ ns _ = ns
```

Both of these cases are trivial. In all future cases, we will assume that r and k are not ns.

Case: $\texttt{joinf}\ r\ k\ (t_1 * \ldots * t_h)$ This case reads

$$\begin{array}{l}
\texttt{| joinf}\ r\ k\ (t_1 * \ldots * t_h)\ \texttt{=} \\
\quad \texttt{let val}\ r_1 * \ldots * r_h\ \texttt{=}\ \texttt{tuplesimp}\ r \\
\quad\quad \texttt{val}\ k_1 * \ldots * k_h\ \texttt{=}\ \texttt{tuplesimp}\ k \\
\quad \texttt{in} \\
\quad\quad \texttt{if}\ \text{for } j \text{ in } 1 \ldots h \text{ we have } (\texttt{joinf}\ r_j\ k_j\ t_j) \neq \text{ns} \\
\quad\quad \texttt{then}\ (\texttt{joinf}\ r_1\ k_1\ t_1) * \ldots * (\texttt{joinf}\ r_h\ k_h\ t_h) \\
\quad\quad \texttt{else ns} \\
\quad \texttt{end}
\end{array}$$

SubCase: $r \leq p$ and $k \leq p$ implies $(\texttt{joinf}\ r\ k\ t_1 * \ldots * t_h) \leq p$ Suppose $r \leq p$ and $k \leq p$. By Lemma 2.22 (Tuple Intersection) on page 40, there are p_1 through p_h such that $p_1 * \ldots * p_h \equiv p$. By TRANS-SUB, Corollary 2.27 (TUPLE-SUB Inversion) on page 45, and soundness of `tuplesimp`, we have

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_j \leq p_j \text{ and } k_j \leq p_j.$$

By induction we can assume that recursive calls to `joinf` are sound, so this implies

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } \texttt{joinf}\ r_j\ k_j\ t_j \preceq p_j \tag{2.54}$$

and then TUPLE-SUB and TRANS-SUB give

$$(\texttt{joinf}\ r_1\ k_1\ t_1) * \ldots * (\texttt{joinf}\ r_h\ k_h\ t_h) \leq p.$$

(2.54) tells us that for j in $1 \ldots h$ we have $(\texttt{joinf}\ r_j\ k_j\ t_j) \neq \text{ns}$, so the definition of `joinf` gives

$$\texttt{joinf}\ r\ k\ (t_1 * \ldots * t_n) \leq p,$$

which is our conclusion.

SubCase: $r \preceq \texttt{joinf}\ r\ k\ t$ By Fact 2.23 (Tuplesimp Sound) on page 41 we know that $r_1 * \ldots * r_h \equiv r$. By soundness of the recursive call to `joinf`,

$$\text{for } j \text{ in } 1 \ldots h \text{ we have } r_j \preceq \texttt{joinf}\ r_j\ k_j\ t_j.$$

If any of the $\texttt{joinf}\ r_j\ k_j\ t_j$'s are $\mathbf{ns}$, then the definition of $\preceq$ gives

$$r \preceq \mathbf{ns}.$$

Otherwise TUPLE-SUB and TRANS-SUB give

$$r \leq (\texttt{joinf}\ r_1\ k_1\ t_1) * \ldots * (\texttt{joinf}\ r_h\ k_h\ t_h).$$

Either way, by definition of this case of `joinf` we have

$$r \preceq \texttt{joinf}\ r\ k\ (t_1 * \ldots * t_n),$$

which is our conclusion.

SubCase: `joinf` terminates. Trivial.

Case: $\texttt{joinf}\ r\ k\ (t_1 \rightarrow t_2)$ The code for this case uses the `ifn` function defined on page 120 to compute the interpretation i of a refinement type. Here it is:

$$\begin{array}{l}
\texttt{| joinf}\ r\ k\ (t_1 \rightarrow t_2)\ \texttt{=} \\
\quad \triangle\{p \rightarrow \texttt{joinf}\ (\texttt{ifn}\ r\ p\ t_1)\ (\texttt{ifn}\ k\ p\ t_1)\ t_2 \mid \\
\qquad p \in \texttt{allrefs}\ t_1 \text{ and} \\
\qquad (\texttt{joinf}\ (\texttt{ifn}\ r\ p\ t_1)\ (\texttt{ifn}\ k\ p\ t_1)\ t_2) \neq \mathbf{ns}\}
\end{array}$$

Before showing that this case of `joinf` works reasonably, we need to show that its interpretation works reasonably. Formally, we will start by showing that if $p \sqsubset t_1$ then

$$i(\texttt{joinf}\ r\ k\ (t_1 \rightarrow t_2))(p) \approx i(r)(p) \sqcup i(k)(p).$$

Suppose p is given and $p \sqsubset t_1$. Consider

$$i(\texttt{joinf}\ r\ k\ (t_1 \rightarrow t_2))(p). \tag{2.55}$$

By definition of `joinf`, this is equal to

$$\begin{array}{l}
i(\triangle\{p' \rightarrow \texttt{joinf}\ (i(r)(p'))\ (i(k)(p'))\ t_2 \mid \\
\qquad p' \in \texttt{allrefs}\ t_1 \text{ and} \\
\qquad (\texttt{joinf}\ (i(r)(p'))\ (i(k)(p'))\ t_2 \neq \mathbf{ns})\})(p)
\end{array}$$

and by definition of i, this is equal to

$$\begin{array}{l}\triangle\{\texttt{joinf}\ \ (i(r)(p'))\ \ (i(k)(p'))\ \ t_2\ \mid \\ \quad p' \in \texttt{allrefs}\ \ t_1 \text{ and} \\ \quad (\texttt{joinf}\ \ (i(r)(p'))\ \ (i(k)(p'))\ \ t_2) \neq \mathbf{ns} \text{ and} \\ \quad p \leq p'\}.\end{array} \tag{2.56}$$

By our induction hypothesis,

$$\texttt{joinf}\ \ (i(r)(p'))\ \ (i(k)(p'))\ \ t_2 \approx i(r)(p') \sqcup i(k)(p').$$

Thus (2.56) is $\approx$ to

$$\begin{array}{l}\triangle\{i(r)(p') \sqcup i(k)(p')\ \mid \\ \quad p' \in \texttt{allrefs}\ \ t_1 \text{ and} \\ \quad (\texttt{joinf}\ \ (i(r)(p'))\ \ (i(k)(p'))\ \ t_2) \neq \mathbf{ns} \text{ and} \\ \quad p \leq p'\}\end{array} \tag{2.57}$$

By Lemma 2.80 (i Monotone in Second Argument) on page 108 and monotonicity of $\sqcup$,

$$p \leq p' \text{ implies } i(r)(p) \sqcup i(k)(p) \leq i(r)(p') \sqcup i(k)(p').$$

Since we can eliminate components from the set in (2.57) that are known to be greater than other components in (2.57) we know that (2.57) is equivalent to

$$\triangle\{i(r)(p) \sqcup i(k)(p)\ \mid (\texttt{joinf}\ \ (i(r)(p))\ \ (i(k)(p))\ \ t_2) \neq \mathbf{ns}\}.$$

By our induction hypothesis, $\texttt{joinf}\ \ (i(r)(p))\ \ (i(k)(p))\ \ t_2$ is $\mathbf{ns}$ if and only if $i(r)(p) \sqcup i(k)(p)$ is $\mathbf{ns}$. Thus this simplifies to

$$i(r)(p) \sqcup i(k)(p). \tag{2.58}$$

Summarizing (2.55) through (2.58), if $p \sqsubset t_1$ then

$$i(\texttt{joinf}\ \ r\ \ k\ \ (t_1 \to t_2))(p) \approx i(r)(p) \sqcup i(k)(p). \tag{2.59}$$

SubCase: $r \preceq p$ and $k \preceq p$ implies $\texttt{joinf}\ \ r\ \ k\ \ (t_1 \to t_2) \preceq p$ Suppose $r \leq p$ and $k \leq p$. By Lemma 2.81 (i Monotone in First Argument) on page 109 we have

$$\text{for all } p' \sqsubset t_1 \text{ we have } i(r)(p') \preceq i(p)(p')$$

and likewise for k gives

$$\text{for all } p' \sqsubset t_1 \text{ we have } i(k)(p') \preceq i(p)(p').$$

Since $\sqcup$ is a least upper bound, this implies

$$\text{for all } p' \sqsubset t_1 \text{ we have } i(r)(p') \sqcup i(k)(p') \preceq i(p)(p'),$$

and (2.59) gives

$$\text{for all } p' \sqsubset t_1 \text{ we have } i(\texttt{joinf}\ r\ k\ (t_1 \to t_2))(p') \preceq i(p)(p'),$$

and Lemma 2.84 (Ordering on i) on page 112 gives

$$\texttt{joinf}\ r\ k\ (t_1 \to t_2) \preceq p,$$

which is our conclusion.

SubCase: $r \preceq \texttt{joinf}\ r\ k\ (t_1 \to t_2)$ Since $\sqcup$ is a least upper bound, we have

$$\text{for all } p \sqsubset t_1 \text{ we have } i(r)(p) \preceq i(r)(p) \sqcup i(k)(p).$$

By (2.59) and Lemma 2.81 (i Monotone in First Argument) on page 109, this implies

$$\text{for all } p \sqsubset t_1 \text{ we have } i(r)(p) \preceq i(\texttt{joinf}\ r\ k\ t_1 \to t_2)(p) \sqcup i(k)(p).$$

By Lemma 2.81 (i Monotone in First Argument) on page 109, this implies

$$r \preceq \texttt{joinf}\ r\ k\ t_1 \to t_2,$$

which is our conclusion.

SubCase: Termination The only loop in this code is over a finite set, and by induction we can assume that the recursive calls to `joinf` terminate.

Case: $\texttt{joinf}\ r\ k\ tc$ The code for this case uses `rconsimp`, which is defined on page 2.6.1. Here is the code:

```
| joinf r k tc =
    let val rc = rconsimp r
        val kc = rconsimp k
    in
        if rc ⊔(def) kc is undefined
        then ns
        else rc ⊔(def) kc
    end
```

SubCase: $r \preceq p$ and $k \preceq p$ implies $\texttt{joinf}\ r\ k\ tc \preceq p$ Suppose $r \leq p$ and $k \leq p$. Then $p \sqsubset tc$, so by Lemma 2.24 (Refinement Constructor Intersection) on page 41 there is a pc such that $p \equiv pc$. Fact 2.25 (Rconsimp Sound) on page 42 gives

$$r \equiv rc$$

and

$$k \equiv kc.$$

Fact 2.29 (RCON-SUB Inversion) on page 45 gives $rc \overset{\text{def}}{\leq} pc$ and $kc \overset{\text{def}}{\leq} pc$. Therefore $rc \overset{\text{def}}{\sqcup} kc$ is defined and

$$rc \overset{\text{def}}{\sqcup} kc \overset{\text{def}}{\leq} pc.$$

By RCON-SUB, TRANS-SUB, and the definition of this case of `joinf`, this implies `joinf` r k tc is not ns and

$$\texttt{joinf}\ r\ \ k\ \ tc \leq p.$$

By definition of $\preceq$, this implies

$$\texttt{joinf}\ r\ \ k\ \ tc \preceq p,$$

which is our conclusion.

SubCase: $r \preceq$ `joinf` r k tc If `joinf` r k tc is ns, then by definition of $\preceq$ we are done.

Otherwise, $rc \overset{\text{def}}{\sqcup} kc$ is defined. By soundness of `rconsimp`, $rc \equiv r$. By properties of $\overset{\text{def}}{\sqcup}$,

$$rc \overset{\text{def}}{\leq} rc \overset{\text{def}}{\sqcup} kc.$$

Using RCON-SUB, TRANS-SUB, and the definition of `joinf`, this implies

$$r \overset{\text{def}}{\leq} \texttt{joinf}\ r\ \ k\ \ tc,$$

which is our conclusion.

SubCase: Termination Trivial. □

We will also define a simple function `sjoinf` that finds the least upper bound of a finite set of generalized refinement types:

```
fun sjoinf t {} = botfn t
  | sjoinf t ({r?} ∪ s?) = joinf r? (sjoinf t s?) t
```

We use `botfn` t as a base case for this recursive function because for all r refining t we have

$$\begin{aligned}(\texttt{botfn}\ t) \sqcup r &= \triangle\{p \in \texttt{allrefs}\ t \mid r \preceq p \text{ and } \texttt{botfn}\ t \preceq p\} \\ &= \triangle\{p \in \texttt{allrefs}\ t \mid r \preceq p\} \\ &\equiv r.\end{aligned}$$

2.10.4 Deciding Refinement Types

In this subsection we will give an algorithm called `infer` and prove that it finds principal refinement types. First in Subsubsection 2.10.4.1 we will give an overview of the algorithm by giving examples of how it works for each case in the syntax. Then in Subsubsection 2.10.4.2 we will give a technical lemma that makes the proof much simpler. Finally in Subsubsection 2.10.4.3 we will describe the algorithm `infer` and prove it correct.

This algorithm is similar to the one actually implemented. The main difference between the algorithm described here and the implementation is the evaluation order; `infer` is eager and the implementation is lazy. For example, when confronted with the expression

```
fn x:bool =>
   case x of
      true => fn _ => false ()
    | false => fn _ => true ()
   end:bool
```

`infer` eagerly finds a type for this by assuming `x` can have all possible refinements of $bool$, yielding the result

$$\bot_{bool} \to \bot_{bool} \wedge tt \to ff \wedge ff \to tt \wedge \top_{bool} \to \top_{bool}.$$

The implementation postpones evaluation as long as possible. It returns a function that, for instance, when passed tt, will return ff. This strategy is faster than eagerness when we are only interested in evaluating functions on a few points in their domain. If pursued in the simplest way, this strategy would be slower than `infer` if we evaluate the function, say, 100 times at tt. The implementation is able to perform well in this case by memoizing. We will discuss the implementation in more detail in Chapter 7.

2.10.4.1 Overview of the Algorithm

In this section we will present the algorithm informally by giving examples of how it behaves for each variety of syntax.

The algorithm has one invariant that needs to be maintained: all types in the environment must have no useful splits. This requires finding a principal split every time we add a variable binding to the environment. The only syntax that adds variable bindings to the environment is abstractions, so all the responsibility for maintaining this invariant falls on that case of the algorithm.

Variable references This case is trivial; just look the variable up in the variable environment. For example, in the environment $[x := tt]$, the type we infer for `x` is tt.

Abstractions Except for the fact that we have to maintain our invariant, we could find a

principal type for `fn x:`$bool$ `=> or (not x, x)` using the following procedure: for each refinement of $bool$, bind `x` to that refinement, and find a principal type for `or (not x, x)` in the resulting environment. Then we would encode the results as an intersection of arrow types; for example, if assuming `x` has the type ff leads to the conclusion `or (not x, x)` has the type tt, then one of the components of the result is $ff \to tt$.

Modifying this procedure to enforce our invariant is fairly simple. Instead of binding `x` to a type r in the environment, find a principal split of r. Bind `x` to each fragment of r and find a principal type for `or (not x, x)` in the resulting environment. Let k be the join of the results; then the component we should add to our result in this case is $r \to k$.

For example, when we consider the refinement $\top_{bool}$ of $bool$, we find its principal split $\{tt, ff\}$. We bind `x` to each of the fragments and find types for `or (not x, x)` in the resulting environments, yielding tt and tt. Then we join these, yielding tt. The final contribution of this reasoning to our result is $\top_{bool} \to tt$. The type for `fn x:`$bool$ `=> or (not x, x)` that results from this procedure is

$$\top_{bool} \to tt \wedge tt \to tt \wedge ff \to tt \wedge \bot_{bool} \to \bot_{bool} .$$

Applications Applications become a call to i at the type level. For example, to find a principal type for `not x`, first we find principal types for `not` and `x`; suppose we get $tt \to ff \wedge ff \to tt \wedge \top_{bool} \to \top_{bool}$ and tt, respectively. Then our result is

$$i(tt \to ff \wedge ff \to tt \wedge \top_{bool} \to \top_{bool})(tt),$$

which is $ff \wedge \top_{bool}$.

Constructor Applications This case is straightforward. Using the $bitstr$ datatype first introduced on page 17 as an example, if we want to find the principal type of an expression like `One (Empty ())`, we first find a principal type for `Empty ()`, yielding *em*. Then our result is the least type rc such that `One` $\overset{\text{def}}{:}$ $em \hookrightarrow rc$; in this case it is *nf*.

By Assumption 2.51 (Constructor And Introduction) on page 67, the least rc such that `One` $\overset{\text{def}}{:}$ $em \hookrightarrow rc$ is the intersection of all of the rc's such that `One` $\overset{\text{def}}{:}$ $em \hookrightarrow rc$. This intersection can be precomputed when the constructors are defined, so this does not affect performance

Case Statements Finding the principal type of a case statement starts as an approximate dual of finding a principal type for constructor applications. First we find a principal type for the expression the case statement is examining; we can use `rconsimp` discussed on page 42 to simplify this type to something of the form rc. For each branch of the case statement with a constructor c we find the set of greatest r's such that $c \overset{\text{def}}{:} r \hookrightarrow rc$. These r's are the possible types of the argument to c that could have given rise to our case object.

The analogy with constructor application is only approximate because we cannot join all of the r's in this set to get a k where $c \stackrel{\text{def}}{:} k \hookrightarrow rc$; in other words, there is no dual to Assumption 2.51 (Constructor And Introduction) on page 67. This is case because $\wedge$ accurately forms intersections of refinement types if we interpret them as sets, but $\sqcup$ only returns an upper bound of the union.

If the set of possible r's for some constructor is empty, then that branch is unreachable. Recall that in the formal language, a branch of a case statement is a function that will be applied to the arguments of the constructor. For each reachable branch e of the case statement we find a principal type for the application of e to some hypothetical value of type r; by the argument we gave for the application case, this is simply a use of i. Since we do not know which of the reachable cases will be taken during execution, we have to take the join of all of these types as the principal type of the case statement.

For example, consider the case statement

```
case Zero (One (Empty ())) of
   Zero => fn arg:bitstr => arg
 | One => fn arg:bitstr => One arg
 | Empty => fn arg:runit => (Empty ())
end:bitstr
```

From the earlier discussion of application, the principal type of `Zero (One Empty)` is nf. The reachable constructors are `Zero` and `One`, where `Zero` has the possible input type nf and `One` has the possible input types nf or em.

The best type for `fn arg:`$bitstr$ `=> arg` applied to a value of type nf is nf, and the best type of `fn arg:`$bitstr$ `=> One arg` applied to a value of type nf or em is nf. Thus the principal type of the case statement is $nf \sqcup nf$, or nf.

Tuples The principal type for a tuple is simply the product of the principal types of the components. For example, to find the principal type for `(not, true ())` we first find the principal types of `not` and `true ()`, yielding $tt \to \mathit{ff} \wedge \mathit{ff} \to tt \wedge \top_{bool} \to \top_{bool}$ and tt respectively. Thus the type for `(not, true ())` is

$$(tt \to \mathit{ff} \wedge \mathit{ff} \to tt \wedge \top_{bool} \to \top_{bool}) * tt.$$

Element Selection To find a principal type of an expression of the form `elt_`m`_`n e, find a principal type for e, simplify it with `tuplesimp`, and then select the appropriate element. For example, a principal type for `(true (), false ())` is $(tt * \top_{bool}) \wedge (\top_{bool} * \mathit{ff})$. Then `tuplesimp` returns $(tt \wedge \top_{bool}) * (\top_{bool} \wedge \mathit{ff})$, so a principal type for `elt_2_2 (true (), false ())` is $\top_{bool} \wedge \mathit{ff}$, which is equivalent to ff.

Fixed Points We find the principal type for a fixed point by iterative approximation. Our

first approximation to the refinement type of the function is the least refinement of its ML type; each successive approximation is the principal type of the body of the fixed point, assuming that recursive references to the function have the previous approximation as their type. When the approximations stop increasing, the last approximation is our principal type.

For example, determining a type for the expression

$$\begin{array}{l}\texttt{fix inc:}\mathit{bitstr} \rightarrow \mathit{bitstr}\ \texttt{=> fn n:}\mathit{bitstr}\ \texttt{=>}\\ \quad\texttt{case n of}\\ \quad\quad\texttt{Empty => fn \_:}\mathit{runit}\ \texttt{=> One (Empty ())}\\ \quad\quad\texttt{| One => fn rest:}\mathit{bitstr}\ \texttt{=> Zero (inc rest)}\\ \quad\quad\texttt{| Zero => fn rest:}\mathit{bitstr}\ \texttt{=> One rest}\\ \quad\texttt{end:}\mathit{bitstr}\end{array}$$

yields these successive approximations to the fixed point:

$$\begin{array}{c}\top_{\mathit{bitstr}} \rightarrow \bot_{\mathit{bitstr}}\\ \bot_{\mathit{bitstr}} \rightarrow \bot_{\mathit{bitstr}} \wedge \mathit{em} \rightarrow \mathit{nf} \wedge \mathit{nf} \rightarrow \mathit{nf} \wedge \top_{\mathit{bitstr}} \rightarrow \top_{\mathit{bitstr}}\\ \bot_{\mathit{bitstr}} \rightarrow \bot_{\mathit{bitstr}} \wedge \mathit{em} \rightarrow \mathit{nf} \wedge \mathit{nf} \rightarrow \mathit{nf} \wedge \top_{\mathit{bitstr}} \rightarrow \top_{\mathit{bitstr}}.\end{array}$$

Since the last two approximations are equivalent, the process terminates and the last approximation is our result.

2.10.4.2 Technical Lemma for Principality

To show that the types from `infer` are principal, we will have to prove

$$\text{if VR} \vdash e : r \text{ then } (\texttt{infer VR } e) \leq r.$$

The premise VR $\vdash e : r$ is awkward to use because the root inference of its derivation may be an inference rule that makes syntactic progress, or it may be WEAKEN-TYPE, AND-INTRO-TYPE, or SPLIT-TYPE. It turns out that it is sufficient to show

$$\text{if VR} \Vdash e : r \text{ then } (\texttt{infer VR } e) \leq r$$

where the root inference of the premise must make syntactic progress. The SPLIT-TYPE case of the proof is most interesting.

Lemma 2.98 (Syntactic Progress Decidability Sufficient) *Let r' be given. If*

$$\textit{for all } r \textit{ we have } \text{VR} \Vdash e : r \textit{ implies } r' \leq r$$

and

$$\text{VR} \vdash e : p$$

and for all x in the domain of VR *we have*

$$\textit{all splits of } \mathrm{VR}(x) \textit{ are useless}$$

then

$$r' \leq p.$$

Proof: By induction on the derivation of $\mathrm{VR} \vdash e : p$. The proof is relatively short because we can trivially handle the cases where the root inference of this derivation makes syntactic progress.

Case: AND-INTRO-TYPE The p has the form $p_1 \wedge p_2$ where the premises of AND-INTRO-TYPE are

$$\mathrm{VR} \vdash e : p_1$$

and

$$\mathrm{VR} \vdash e : p_2.$$

Two uses of the induction hypothesis give

$$r' \leq p_1$$

and

$$r' \leq p_2.$$

Then AND-INTRO-SUB gives

$$r' \leq p_1 \wedge p_2,$$

which is our conclusion.

Case: WEAKEN-TYPE Then the premises of WEAKEN-TYPE are

$$\mathrm{VR} \vdash e : p'$$

and

$$p' \leq p.$$

By induction hypothesis,

$$r' \leq p',$$

so TRANS-SUB gives

$$r' \leq p$$

which is our conclusion.

Case: SPLIT-TYPE Then VR has the form $\mathrm{VR}'[y := k]$ where the premises of `split-type` are

$$k \asymp s$$

and

$$\text{for } k' \text{ in } s \text{ we have } \mathrm{VR}'[y := k'] \vdash e : p.$$

By hypothesis, s is a useless split of k, so we can choose a k' in s such that $k \equiv k'$.

We will be putting k' in an environment and then using the induction hypothesis, so we need to know that all splits of k' are useless. To show this, suppose that $k' \asymp s'$. Then EQUIV-SPLIT-L gives $k \asymp s'$, and by hypothesis there is therefore a k'' in s' such that $k \equiv k''$. TRANS-SUB then gives $k' \equiv k''$. Thus,

$$\text{all splits of } k' \text{ are useless.}$$

Now we have to take cases on the form of e to show that

$$\mathrm{VR}'[y := k'] \Vdash e : r \text{ implies } r' \leq r.$$

The most interesting of the following cases is when e is some variable other than y.

SubCase: e is not a variable Suppose that

$$\mathrm{VR}'[y := k'] \Vdash e : r.$$

Lemma 2.66 (Environment Modification) on page 81 gives

$$\mathrm{VR}'[y := k] \Vdash e : r$$

and our hypothesis gives

$$r' \leq r.$$

Summarizing, the reasoning so far in this subcase gives

$$\mathrm{VR}'[y := k'] \Vdash e : r \text{ implies } r' \leq r.$$

SubCase: e has the form y Suppose

$$\mathrm{VR}'[y := k'] \Vdash y : r.$$

The last inference of this must be VAR-TYPE, so $r = k'$. VAR-TYPE gives

$$\mathrm{VR}'[y := k] \Vdash y : k,$$

so our hypothesis gives

$$r' \leq k.$$

Since $k \equiv k'$, this implies $r' \leq r$. Summarizing the steps in this subcase so far,

$$\mathrm{VR}'[y := k'] \Vdash y : r \text{ implies } r' \leq r.$$

SubCase: e has the form z, where $z \neq y$ Suppose

$$\text{VR}'[y := k'] \Vdash z : r.$$

Since $z \neq y$, this derivation ignores y, so we also have

$$\text{VR}'[y := k] \Vdash z : r.$$

By hypothesis, this implies

$$r' \leq r.$$

Summarizing the steps in this subcase so far,

$$\text{VR}'[y := k'] \Vdash z : r \text{ implies } r' \leq r.$$

Regardless of which subcase we use, one of the premises of SPLIT-TYPE is

$$\text{VR}'[y := k'] \vdash e : p.$$

We can use the induction hypothesis on this and the result from whichever subcase we used to get

$$r' \leq p,$$

which is our conclusion.

Case: Any rule that makes syntactic progress In that case we have

$$\text{VR} \Vdash e : p$$

so our hypothesis gives

$$r' \leq p,$$

which is our conclusion. □

2.10.4.3 Definition and Proof of Refinement Type Inference Algorithm

The decision procedure for monomorphic refinement types is in Figures 2.7 and 2.8. This procedure takes an expression and an environment mapping variables to refinement types, and it returns a principal refinement type for the expression if there is one, or ns otherwise. It has three interesting properties: it always terminates, it returns a refinement type for the given expression, and the type is principal. We will prove one of these properties in each of the next three theorems. The most interesting theorem in this Subsubsection is Theorem 2.101 (Infer Returns Principal Type) on page 151; the most interesting cases of each theorem deal with `case` and `fix` statements.

The portions of the algorithm that deal with `case` and `fix` statements use the "`as`" keyword. This has not appeared so far in this thesis. It simultaneously binds a variable to a structure and other variables to the parts of the structure; for example, binding the pattern `x as (y, z)` to the value `(true, false)` binds `y` to `true`, `z` to `false`, and `x` to `(true, false)`.

```
fun infer VR y =
   if for some t we have VR(y) ⊏ t
   then VR(y)
   else ns
| infer VR (fn x:t => e') =
     if there is a u such that rtom(VR)[x := t] ⊢ e' :: u
     then
        let val u = the unique u such that rtom(VR)[x := t] ⊢ e' :: u
            fun do_one r =
               sjoinf u {infer (VR[x := r']) e' | r' ∈ split r}
         in
            △fn {r → do_one r | r ∈ allrefs t and (do_one r) ≠ ns}
         end
     else ns
| infer VR (e1 e2) =
     let val r? = infer VR e1
         val k? = infer VR e2
     in
         if r? = ns or k? = ns
         then ns
         else
            let
               val u → t = rtom(r?)
               val u' = rtom(k?)
            in
               if u = u'
               then ifn r? k? u
               else ns
            end
     end
| infer VR (c e') =
     let val k? = infer VR e'
         val t = the unique t such that c ::def t ↪ tc
     in
         △fn {rc | r ∈ allrefs t and subtypep k? r t and c :def r ↪ rc}
     end
```

Figure 2.7: Decision Procedure for Refinement Types Part 1

```
| infer VR (e as (case e_0 of c_1 => e_1 | ... | c_n => e_n end:t)) =
    if not rtom(VR) ⊢ e :: t then ns
    else let val r? = infer VR e_0
         in if r? = ns then ns
            else let val rc = rconsimp (r?)
                 in
                     sjoinf t {ifn (infer VR e_h) p u |
                                   h ∈ 1..n
                               and c_h ::(def) u ↪ uc
                               and p ∈ allrefs u
                               and c_h :(def) p ↪ rc}
                 end
         end
| infer VR (e_1, ..., e_n) =
    if for any i in 1..n we have infer VR e_i = ns
    then ns
    else infer VR e_1 * ... * infer VR e_n
| infer VR (elt_m_n e') =
    let val k? = infer VR e'
    in
       if k? = ns then ns
       else let val k_1 * ... * k_n = tuplesimp (k?)
            in k_m end
    end
| infer VR (e as (fix f:t => fn x:t_1 => e')) =
    let fun loop r =
            let val next? = infer (VR[f := r]) (fn x:t_1 => e')
            in
              if subtypep next? r t then r
              else if next? = ns then ns
              else loop next?
            end
        val t'_1 → t_2 = t
    in
       if t'_1 ≠ t_1 then ns
       else if ML type inference does not give rtom(VR) ⊢ e :: t_1 → t_2
       then ns
       else loop (botfn (t_1 → t_2))
    end
```

Figure 2.8: Decision Procedure for Refinement Types Part 2

According to Theorem 2.54 (Inferred Types Refine) on page 68, the refinement type inference rules in Figure 2.6 ensure that the refinement type environment gives a well-formed refinement type for each free variable in the expression and that the expression has an ML type. Since `infer` is an implementation of these rules, it does the same. The alternative would be to assume we only use `infer` on terms and environments that are consistent with ML typing. The extra hypothesis would complicate the proofs and obscure the correspondence between the refinement type inference rules and the algorithm, so the approach used below seems best.

The algorithm has an invariant: the refinement types in the environment must have no useful splits. Because of this assumption, we never need to consider using the SPLIT-TYPE rule to split variables that are in the initial environment. As we execute the algorithm, we maintain the invariant by taking the principal split of the type of each new variable before we add it to the environment. The discussion of principal splits above should make it intuitively clear that this is appropriate; for a formal justification, see the cases for abstractions and fixed points in the proof of Theorem 2.101 (Infer Returns Principal Type) on page 151.

We start with a simple lemma saying that the argument of the tail recursive loop in the `fix` case of `infer` always refines the same ML type. The hypothesis of the lemma is always true since it is implied by Theorem 2.100 (Infer Returns Some Type) on page 145; the hypothesis saves us from having a lemma nested inside a theorem. Note that the variables t_1 and t_2 mentioned in the lemma are defined in the `fix` case of `infer`.

Lemma 2.99 (Fix Case of Infer is Well-Behaved) *In the* `fix` *case of* `infer`*, we will abbreviate* `fn` $x{:}t_1$ `=>` e' *as* e''*. If, for all* r*,*

$$\texttt{infer}\ (\mathrm{VR}[f := r])\ e'' \textit{ is not } \textbf{ns}$$

implies

$$\mathrm{VR}[f := r] \vdash e'' : \texttt{infer}\ (\mathrm{VR}[f := r])\ e'',$$

then the argument of `loop` *always refines* $t_1 \to t_2$*.*

Proof: By induction on the evaluation `loop`.

Case: Initial call to `loop` This is trivial, since the argument to the initial call of `loop` is `botfn` $(t_1 \to t_2)$, which obviously refines $t_1 \to t_2$.

Case: Recursive calls to `loop` We can assume the incoming argument r of `loop` refines $t_1 \to t_2$, and we need to show that the value *next?* that will be passed to the next recursive call also refines $t_1 \to t_2$. Since $next? = \texttt{infer}\ (\mathrm{VR}[f := r])\ (\texttt{fn}\ x{:}t_1\ \texttt{=>}\ e')$, our hypothesis gives

$$\mathrm{VR}[f := r] \vdash e'' : next?.$$

Theorem 2.54 (Inferred Types Refine) on page 68 then gives a t' such that $next? \sqsubset t'$ and

$$\mathrm{rtom}(\mathrm{VR}[f := r]) \vdash e'' :: t'.$$

Since $r \sqsubset t_1 \to t_2$,

$$\text{rtom}(\text{VR}[f := r]) = \text{rtom}(\text{VR})[f := t_1 \to t_2].$$

If we ever call `loop` then t must have the form $t_1 \to t_2$, and we must also have

$$\text{rtom}(\text{VR}) \vdash e :: t_1 \to t_2.$$

The last inference of this must be FIX-VALID with the premise

$$\text{rtom}(\text{VR})[f := t_1 \to t_2] \vdash e'' :: t_1 \to t_2$$

Thus Lemma 2.4 (Unique Inferred ML Types) on page 27 gives $t' = t_1 \to t_2$. Since $next? \sqsubset t'$, this is our conclusion. □

In the next theorem we have the hypothesis "`infer` VR `e` terminates". By Theorem 2.102 (Infer Terminates) on page 160, this is always true. Once again we are using these always true hypotheses to break up the decidability proof into manageable chunks.

Theorem 2.100 (Infer Returns Some Type)
If `infer` VR e *terminates and* `infer` VR e *is not* **ns** *then*

$$\text{VR} \vdash e : \texttt{infer VR } e.$$

Proof: By induction on e.

Case: $e = y$ The code for this case is

```
fun infer VR y =
    if for some t we have VR(y) ⊏ t
    then VR(y)
    else ns
```

Since `infer` VR e is not ns, it is VR(e) and for some t we have VR$(e) \sqsubset t$. Thus VAR-TYPE gives VR $\vdash e :$ `infer` VR e, which is our conclusion.

Case: $e =$ `fn` x`:`t `=>` e' The code for this case is

```
| infer VR (fn x:t => e') =
    if there is a u such that rtom(VR)[x := t] ⊢ e' :: u
    then
       let val u = the unique u such that rtom(VR)[x := t] ⊢ e' :: u
           fun do_one r =
              sjoinf u {infer (VR[x := r']) e' | r' ∈ split r}
        in
           △fn {r → do_one r | r ∈ allrefs t and (do_one r) ≠ ns}
        end
    else ns
```

Since `infer` VR e is not ns, there is a u such that rtom(VR) $\vdash e' :: u$. By Lemma 2.4 (Unique Inferred ML Types) on page 27, there is exactly one such u.

Since `infer` VR e terminates, soundness of $\triangle$`fn` tells us that all calls to `do_one` terminate. Since `infer` VR e is not ns, at least one of the calls to `do_one` returns a refinement type instead of ns. Suppose `do_one` r does not return ns; by definition of `do_one`, this implies

for all r' in `split` r we have `infer` (VR$[x := r']$) e' terminates and is not ns.

We can use our induction hypothesis to get

for all r' in `split` r we have VR$[x := r'] \vdash e'$: `infer` (VR$[x := r']$) e'

By WEAKEN-TYPE and soundness of `sjoinf`, we get

for all r' in `split` r we have VR$[x := r'] \vdash e'$: `do_one` r.

Soundness of `split` tells us $r \asymp$ `split` r. Thus SPLIT-TYPE gives

VR$[x := r] \vdash e'$: `do_one` r.

Then ABS-TYPE gives

VR $\vdash$ `fn` x:t `=>` e' : $r \to$ `do_one` r.

This is true for all r refining t for which `do_one` r is not ns, so repeated use of AND-INTRO-TYPE gives

VR $\vdash$ `fn` x:t `=>` e' : `infer` VR e,

which is our conclusion.

Case: $e = e_1\ e_2$ The code for this case is

```
| infer VR (e1 e2) =
    let val r? = infer VR e1
        val k? = infer VR e2
    in
        if r? = ns or k? = ns
        then ns
        else
           let
              val u → t = rtom(r?)
              val u' = rtom(k?)
           in
              if u = u'
              then ifn r? k? u
              else ns
           end
    end
```

Since `infer` VR e is not ns, both $r?$ and $k?$ must not be ns. Call them r and k respectively. Since $r =$ `infer` VR e_1 and $k =$ `infer` VR e_2, the induction hypothesis gives

$$\text{VR} \vdash e_1 : r$$

and

$$\text{VR} \vdash e_2 : k.$$

Since $r \sqsubset u \to t$, we know that r has the form $r_1 \to r_1' \wedge \ldots \wedge r_n \to r_n'$. By soundness of `ifn`,

$$\texttt{ifn}\ r\ \ k\ \ u = \wedge\{r_j' \mid j \in 1 \ldots n \text{ and } k \leq r_j\}.$$

Since VR $\vdash e_2 : k$, WEAKEN-TYPE gives

$$\text{for } j \text{ in } 1 \ldots n \text{ such that } k \leq r_j \text{ we have VR} \vdash e_2 : r_j.$$

Since VR $\vdash e_1 : r$, we can use WEAKEN-TYPE to get

$$\text{for all } j \text{ in } 1 \ldots n \text{ we have VR} \vdash e_1 : r_j \to r_j'.$$

Therefore APPL-TYPE gives

$$\text{for } j \text{ in } 1 \ldots n \text{ such that } k \leq r_j \text{ we have VR} \vdash e_1\ \ e_2 : r_j'$$

and repeated use of AND-INTRO-TYPE gives

$$\text{VR} \vdash e_1\ \ e_2 : \texttt{ifn}\ r\ \ k\ \ u.$$

By definition of this case of `infer`, this is our conclusion.

Case: $e = c\ \ e'$ The code for this case is

```
| infer VR (c e') =
    let val k? = infer VR e'
        val t = the unique t such that c ::def t ↪ tc
    in
        △fn {rc | r ∈ allrefs t and subtypep k? r t and c :def r ↪ rc}
    end
```

Since `infer` VR e is not ns, soundness of $\triangle$`fn` and `subtypep` tell us that $k?$ cannot be ns. Thus we will call it k. The value of `infer` VR e must have the form

$$rc_1 \wedge \ldots \wedge rc_n$$

where for i in $1 \ldots n$ we have an r_i refining t such that $c :^{\text{def}} r_i \hookrightarrow rc_i$ and $k \leq r_i$. By induction hypothesis,

$$\text{VR} \vdash e' : k,$$

and by WEAKEN-TYPE,

$$\text{for } i \text{ in } 1 \ldots n \text{ we have VR} \vdash e' : r_i.$$

Then CONSTR-TYPE gives

$$\text{for } i \text{ in } 1 \ldots n \text{ we have VR} \vdash c\ \ e' : rc_i$$

and repeated use of AND-INTRO-TYPE gives

$$\text{VR} \vdash c\ \ e' : rc_1 \wedge \ldots \wedge rc_n,$$

which is our conclusion.

Case: $e = \texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}t$ The code for this case is

```
| infer VR (e as (case e0 of c1 => e1 | ... | cn => en end:t)) =
    if not rtom(VR) ⊢ e :: t then ns
    else let val r? = infer VR e0
         in if r? = ns then ns
            else let val rc = rconsimp (r?)
                 in
                     sjoinf t {ifn (infer VR eh) p u |
                                   h ∈ 1..n
                               and ch :def: u ↪ uc
                               and p ∈ allrefs u
                               and ch :def: p ↪ rc}
                 end
         end
```

Let k be the result of this case of `infer`. By hypothesis this case does not return ns, so `infer` VR e_0 is defined. Our induction hypothesis, soundness of `rconsimp`, and WEAKEN-TYPE give

$$\text{VR} \vdash e_0 : rc.$$

Let h in $1 \ldots n$, u, and $p' \sqsubset u$ be given such that

$$c_h :^{\text{def}} p' \hookrightarrow rc. \tag{2.60}$$

By soundness of `allrefs`, there is a p in `allrefs` u such that $p \equiv p'$, and Assumption 2.52 (Constructor Argument Strengthen) on page 67 gives

$$c_h :^{\text{def}} p \hookrightarrow rc.$$

Then, by soundness of `sjoinf`, we have

$$\texttt{ifn (infer VR}\ e_h\texttt{)}\ p\ u \leq k.$$

By soundness of `ifn` and Lemma 2.83 (i Gives an Upper Bound) on page 111, this implies

$$\texttt{infer VR } e_h \leq p \to k.$$

By induction hypothesis,

$$\text{VR} \vdash e_h : \texttt{infer VR } e_h,$$

and then WEAKEN-TYPE gives

$$\text{VR} \vdash e_h : p \to k.$$

Since $p \equiv p'$, we have $p \to k \equiv p' \to k$, so using WEAKEN-TYPE again gives

$$\text{VR} \vdash e_h : p' \to k.$$

Summarizing the argument from (2.60) to here,

$$\begin{array}{l}\text{for all } h \text{ in } 1 \ldots n, \\ \quad c_h \overset{\text{def}}{:} p' \hookrightarrow rc \\ \text{implies} \\ \quad \text{VR} \vdash e_h : p' \to k.\end{array}$$

The algorithm explicitly ensures that

$$\text{rtom(VR)} \vdash e :: t.$$

Since `sjoinf` t s always returns a refinement of t,

$$k \sqsubset t.$$

Thus we can use CASE-TYPE to get

$$\text{VR} \vdash e : k,$$

which is our conclusion.

Case: $e = (e_1, \ldots, e_n)$ The code for this case is

```
| infer VR (e_1, ..., e_n) =
    if for any i in 1..n we have infer VR e_i = ns
    then ns
    else infer VR e_1 * ... * infer VR e_n
```

Since `infer VR` e is not `ns`, for h in $1 \ldots n$ we have `infer VR` e_h is not `ns`. By induction hypothesis, this implies

$$\text{for } h \text{ in } 1 \ldots n \text{ we have VR} \vdash e_h : \texttt{infer VR } e_h$$

and then TUPLE-TYPE gives

$$\text{VR} \vdash (e_1, \ldots, e_n) : \texttt{infer VR } e_1 * \ldots * \texttt{infer VR } e_n$$

which is our conclusion.

Case: $e =$ `elt_`m`_`n e' The code for this case is

```
  | infer VR (elt_m_n e') =
      let val k? = infer VR e'
      in
         if k? = ns then ns
         else let val k1 * ... * kn = tuplesimp (k?)
              in km end
      end
```

Since `infer VR` e is not ns, `infer VR` e' must not be ns; call it k. By induction hypothesis we must have

$$\text{VR} \vdash e' : k.$$

By soundness of `tuplesimp`, $k_1 * \ldots * k_n \equiv k$, and by WEAKEN-TYPE, $\text{VR} \vdash e' : k_1 * \ldots * k_n$. Then ELT-TYPE gives $\text{VR} \vdash$ `elt_`m`_`n $e' : k_m$, which is our conclusion.

Case: $e =$ `fix` $f{:}t$ `=>` `fn` $x{:}t_1$ `=>` e' The code for this case is

```
| infer VR (e as (fix f:t => fn x:t1 => e')) =
    let fun loop r =
            let val next? = infer (VR[f := r]) (fn x:t1 => e')
            in
              if subtypep next? r t then r
              else if next? = ns then ns
              else loop next?
            end
        val t1' -> t2 = t
    in
       if t1' ≠ t1 then ns
       else if ML type inference does not give rtom(VR) ⊢ e :: t1 -> t2
       then ns
       else loop (botfn (t1 -> t2))
    end
```

We will abbreviate `fn` $x{:}t_1$ `=>` e' as e''. Suppose `infer VR` e returns r. The definition of `loop` tells us that $next? \preceq r$ where

$$next? = \texttt{infer}\ (\text{VR}[f := r])\ e''.$$

Our induction hypothesis gives

$$\text{VR}[f := r] \vdash e'' : next?$$

and WEAKEN-TYPE used with $next? \preceq r$ gives

$$\text{VR}[f := r] \vdash e'' : r.$$

Lemma 2.99 (Fix Case of Infer is Well-Behaved) on page 144 gives $r \sqsubset t_1 \to t_2$, so FIX-TYPE gives

$$\text{VR} \vdash \texttt{fix}\ \ f : t_1 \to t_2\ \ \texttt{=>}\ \ e'' : r$$

which is our conclusion. □

The next theorem shows that when `infer` returns a refinement type, it returns a principal refinement type. One of the hypotheses is that `infer` terminates on its input. Theorem 2.102 (Infer Terminates) on page 160 tells us this is always true, but we have to have an explicit hypothesis here because we have not yet proved that theorem. An alternative would be to prove both theorems at once; that would lead to one large proof instead of two smaller ones.

Theorem 2.101 (Infer Returns Principal Type) *If*

all splits of types in VR *are useless*

and

`infer` VR e *terminates*

then

if there is an r *such that* $\text{VR} \vdash e : r$ *then*
$(\texttt{infer}\ \ \text{VR}\ \ e) \preceq r$.

Proof: By induction on e. But first we need to derive the simple consequence of Lemma 2.98 (Syntactic Progress Decidability Sufficient) on page 138 that we will use to prove principality:

Suppose

$$\begin{array}{c}\text{VR} \Vdash e : r \text{ implies} \\ (\texttt{infer}\ \ \text{VR}\ \ e) \preceq r.\end{array} \tag{2.61}$$

Also suppose

$$\text{VR} \vdash e : r. \tag{2.62}$$

Clearly any derivation of (2.62) is going to include a derivation of $\text{VR} \Vdash e : r'$ for some r'. Therefore, (2.61) gives

$$\texttt{infer}\ \ \text{VR}\ \ e \text{ is not ns} \tag{2.63}$$

and Lemma 2.98 (Syntactic Progress Decidability Sufficient) on page 138 gives

$$(\texttt{infer}\ \ \text{VR}\ \ e) \leq r. \tag{2.64}$$

Thus (2.62) implies (2.63) and (2.64), which is our principality result. The reasoning so far tells us that (2.61) implies principality:

$$\begin{array}{l}\text{For all } r,\\ (\text{VR} \Vdash e : r \text{ implies}\\ \quad (\texttt{infer VR } e) \preceq r)\\ \text{implies}\\ \text{For all } r,\\ (\text{VR} \vdash e : r \text{ implies}\\ \quad (\texttt{infer VR } e) \preceq r)\end{array} \tag{2.65}$$

In each case of the proof below, we shall use (2.65) to establish principality instead of doing it directly.

Case: $e = y$ The code for this case is

```
fun infer VR y =
    if for some t we have VR(y) ⊏ t
    then VR(y)
    else ns
```

Suppose $\text{VR} \Vdash y : r$. Then the last inference of this must be VAR-TYPE with the premises $\text{VR}(y) = r$ and $r \sqsubset t$. Since $\text{VR}(y) \sqsubset t$, we know that `infer VR y` returns $\text{VR}(y)$. By SELF-SUB, this implies $(\texttt{infer VR } e) \leq r$, which in turn implies $(\texttt{infer VR } e) \preceq r$. Thus (2.65) gives principality.

Case: $e = \texttt{fn } x\texttt{:}t \texttt{ => } e'$ The code for this case is

```
| infer VR (fn x:t => e') =
    if there is a u such that rtom(VR)[x := t] ⊢ e' :: u
    then
      let val u = the unique u such that rtom(VR)[x := t] ⊢ e' :: u
          fun do_one r =
              sjoinf u {infer (VR[x := r']) e' | r' ∈ split r}
       in
          △fn {r → do_one r | r ∈ allrefs t and (do_one r) ≠ ns}
       end
    else ns
```

Suppose

$$\text{VR} \Vdash (\texttt{fn } x\texttt{:}t \texttt{ => } e') : p.$$

The last inference of this must be ABS-TYPE, where p has the form $k'' \to k'$ and the premises of ABS-TYPE are

$$k'' \sqsubset t$$

and

$$\text{VR}[x := k''] \vdash e' : k'.$$

By soundness of `allrefs`, there is a k in `allrefs` t such that $k \equiv k''$. Suppose r' is in `split` k. Then $r' \leq k$, so $r' \leq k''$ and Lemma 2.66 (Environment Modification) on page 81 gives

$$\text{VR}[x := r'] \vdash e' : k'.$$

Because `split` is sound, r' has no useful splits. Thus we can use our induction hypothesis to get

$$(\texttt{infer}\ (\text{VR}[x := r'])\ e') \preceq k'.$$

Because `sjoinf` is sound and $\sqcup$ is a least upper bound,

$$\texttt{do\_one}\ k \preceq k'.$$

Thus ARROW-SUB gives

$$k \to \texttt{do\_one}\ k \leq k \to k'.$$

Since $k \equiv k''$, we can use TRANS-SUB and RCON-SUB to get

$$k \to \texttt{do\_one}\ k \leq k'' \to k'.$$

Thus the definition of `infer` and repeated use of AND-ELIM-L-SUB and AND-ELIM-R-SUB give

$$(\texttt{infer}\ \text{VR}\ e) \leq k'' \to k'.$$

Summarizing the argument so far in this subcase,

$$\text{VR} \Vdash \texttt{fn}\ x : t\ \texttt{=>}\ e' : p \text{ implies}$$
$$(\texttt{infer}\ \text{VR}\ e) \preceq p.$$

By (2.65), this implies our conclusion.

Case: $e = e_1\ e_2$ The code for this case is

```
| infer VR (e1 e2) =
    let val r? = infer VR e1
        val k? = infer VR e2
    in
        if r? = ns or k? = ns
        then ns
        else
           let
              val u → t = rtom(r?)
              val u' = rtom(k?)
           in
              if u = u'
              then ifn r? k? u
              else ns
           end
    end
```

Suppose VR $\Vdash e_1\ e_2 : p'$. The only way to infer this is with APPL-TYPE with the premises

$$\text{VR} \vdash e_1 : p \to p'$$

and

$$\text{VR} \vdash e_2 : p.$$

Since `infer` VR e terminates, both `infer` VR e_1 and `infer` VR e_2 must terminate. Thus we can use the induction hypothesis on each of these to get

$$\texttt{infer}\ \text{VR}\ e_1 \preceq p \to p'$$

and

$$\texttt{infer}\ \text{VR}\ e_2 \preceq p.$$

Abbreviate `infer` VR e_1 as r and `infer` VR e_2 as k.

Uninteresting reasoning about refinement types tells us that rtom(r) will indeed have the form $u \to t$ and that rtom$(k) = u$.

By definition of i we have $i(p \to p')(p) = p'$. Lemma 2.81 (i Monotone in First Argument) on page 109 implies $i(r)(p) \preceq i(p \to p')(p)$, and Lemma 2.80 ($i$ Monotone in Second Argument) on page 108 implies $i(r)(k) \preceq i(r)(p)$. Thus

$$i(r)(k) \preceq p'. \tag{2.66}$$

Thus `infer` VR e is not ns. We can use the definition of `infer` to rewrite (2.66), yielding

$$\texttt{infer}\ \text{VR}\ e \preceq p'.$$

The argument in this subcase so far can be summarized as

$$\text{VR} \Vdash e_1 \;\; e_2 : p' \text{ implies}$$
$$\texttt{infer VR } e \preceq p'.$$

By (2.65), this implies our conclusion.

Case: $e = c \;\; e'$ The code for this case is

```
| infer VR (c e') =
    let val k? = infer VR e'
        val t = the unique t such that c ::def t ↪ tc
    in
        △fn {rc | r ∈ allrefs t and subtypep k? r t and c :def r ↪ rc}
    end
```

Suppose

$$\text{VR} \Vdash c \;\; e' : p.$$

The last inference of this must be CONSTR-TYPE, where p has the form pc and the premises of CONSTR-TYPE are

$$c :^{\text{def}} p' \hookrightarrow pc$$

and

$$\text{VR} \vdash e' : p'.$$

Our induction hypothesis gives

$$k? \preceq p'.$$

By Assumption 2.2 (Constructors have Unique ML Types) on page 26, there are unique t and tc such that $c ::^{\text{def}} t \hookrightarrow tc$. By Assumption 2.49 (Constructor Type Refines) on page 65, $p' \sqsubset t$, so there is an r in `allrefs` t such that $p' \equiv r$. Then Assumption 2.52 (Constructor Argument Strengthen) on page 67 gives

$$c :^{\text{def}} r \hookrightarrow pc$$

and TRANS-SUB gives

$$k? \preceq r.$$

Thus $pc \in \{rc \mid r \in \texttt{allrefs } t \text{ and } \texttt{subtypep } k? \;\; r \;\; t \text{ and } c :^{\text{def}} r \hookrightarrow rc\}$.

Since this set is not empty, this call to `infer` does not return ns. By repeated use of AND-ELIM-L-SUB and AND-ELIM-R-SUB,

$$(\triangle\{rc \mid r \in \texttt{allrefs } t \text{ and } \texttt{subtypep } k? \;\; r \;\; t \text{ and } c :^{\text{def}} r \hookrightarrow rc\}) \leq pc.$$

Summarizing the argument so far in this subcase,

$$\text{VR} \Vdash e : p \text{ implies}$$
$$(\texttt{infer VR } e) \preceq p.$$

By (2.65), this implies our conclusion.

Case: $e = \texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}t$ The code for this case is

```
| infer VR (e as (case e0 of c1 => e1 | ... | cn => en end:t)) =
    if not rtom(VR) ⊢ e :: t then ns
    else let val r? = infer VR e0
         in if r? = ns then ns
            else let val rc = rconsimp (r?)
                 in
                     sjoinf t {ifn (infer VR eh) p u |
                                   h ∈ 1..n
                               and ch :def: u ↪ uc
                               and p ∈ allrefs u
                               and ch :def: p ↪ rc}
                 end
         end
```

Suppose we have an r such that $\text{VR} \Vdash e : r$. The last inference of this must be CASE-TYPE with the premises

$$\begin{gathered}\text{VR} \vdash e_0 : kc, \\ r \sqsubset t, \\ \text{for } h \text{ in } 1 \ldots n \text{ and all } k \text{ such that} \\ c_h \overset{\text{def}}{:} k \hookrightarrow kc \\ \text{we have} \\ \text{VR} \vdash e_h : k \to r,\end{gathered} \tag{2.67}$$

and

$$\text{rtom}(\text{VR}) \vdash e :: t.$$

Thus the ML type checking in this case of `infer` succeeds and this case of `infer` evaluates `infer` VR e_0. Since this case of `infer` terminates, `infer` VR e_0 must terminate. Thus our induction hypothesis gives $(\texttt{infer}\ \text{VR}\ e_0) \preceq kc$. Let $rc = \texttt{rconsimp}\ (\texttt{infer}\ \text{VR}\ e_0)$; soundness of `rconsimp` then gives $rc \leq kc$.

Now we shall show that for all h in $1 \ldots n$ and all p such that $c_h \overset{\text{def}}{:} p \hookrightarrow rc$,

$$i(\texttt{infer}\ \text{VR}\ e_h)(p) \leq r.$$

First choose h in $1 \ldots n$ and p such that $c_h \overset{\text{def}}{:} p \hookrightarrow rc$. By (2.67), this implies $\text{VR} \vdash e_h : p \to r$. Then the induction hypothesis gives

$$\texttt{infer}\ \text{VR}\ e_h \preceq p \to r.$$

By Corollary 2.82 (Bound on Argument to i Gives Bound on i) on page 111, this implies

$$i(\texttt{infer VR } e_h)(p) \leq r.$$

Since this holds for all h in $1 \ldots n$ and all p such that $c_h \stackrel{\text{def}}{:} p \hookrightarrow rc$, the call to `sjoinf` in this case of `infer` does not return ns. Thus

$$\texttt{infer VR } e \text{ is not ns}$$

and `sjoinf` computes a least upper bound, so

$$(\texttt{infer VR } e) \preceq r.$$

Summarizing the argument so far,

$$\begin{array}{l}\text{VR} \Vdash e : r \text{ implies} \\ \quad (\texttt{infer VR } e) \preceq r.\end{array}$$

By (2.65), this implies principality.

Case: $e = (e_1, \ldots, e_n)$ The code for this case is

```
| infer VR (e_1, ..., e_n) =
    if for any i in 1..n we have infer VR e_i = ns
    then ns
    else infer VR e_1 * ... * infer VR e_n
```

Suppose VR$\Vdash (e_1, \ldots, e_n) : r$. The last inference of this must be TUPLE-TYPE, so r has the form $r_1 * \ldots * r_n$ and the premises of TUPLE-TYPE are

$$\text{for } h \text{ in } 1 \ldots n \text{ we have VR} \vdash e_h : r_h.$$

Our induction hypothesis gives

$$\text{for } h \text{ in } 1 \ldots n \text{ we have } (\texttt{infer VR } e_h) \preceq r_h.$$

This immediately tells us that `infer VR` e is not ns. RCON-SUB gives

$$\texttt{infer VR } e_1 * \ldots * \texttt{infer VR } e_n \leq r_1 * \ldots * r_n$$

and by definition of this case of `infer`, this is equivalent to

$$\texttt{infer VR } e \leq r.$$

Summarizing the argument so far,

$$\begin{array}{l}\text{VR} \Vdash e : r \text{ implies} \\ \quad (\texttt{infer VR } e) \preceq r.\end{array}$$

By (2.65), this implies principality.

Case: $e =$ `elt_`m`_`n e' The code for this case is

```
| infer VR (elt_m_n e') =
    let val k? = infer VR e'
    in
       if k? = ns then ns
       else let val k1 * ... * kn = tuplesimp (k?)
            in km end
    end
```

Suppose VR $\Vdash$ `elt_`m`_`n $e' : r$. The last inference of this must be ELT-TYPE with the premise VR $\vdash e' : r_1 * \ldots * r_n$ where $r = r_m$. By induction hypothesis, `infer` VR e' is not ns; call it k. The induction hypothesis also gives $k \leq r_1 * \ldots * r_n$. By Theorem 2.21 (Subtypes Refine) on page 36, there must be a t such that $k \sqsubset t$ and $r_1 * \ldots * r_n \sqsubset t$. We can only have $r_1 * \ldots * r_n \sqsubset t$ if t has the form $t_1 * \ldots * t_n$. Thus k is a valid input to `tuplesimp`, and soundness of `tuplesimp` gives $k \equiv$ `tuplesimp` k.

Let $k_1 * \ldots * k_m =$ `tuplesimp` k. Then TRANS-SUB gives $k_1 * \ldots * k_m \leq r_1 * \ldots * r_n$, and Corollary 2.27 (TUPLE-SUB Inversion) on page 45 gives $k_m \leq r_m$. But $k_m =$ `infer` VR e and $r_m = r$, so we have (`infer` VR e) $\leq r$.

Summarizing the argument so far,

VR $\Vdash$ `elt_`m`_`n $e' : r$ implies
`infer` VR $e \preceq r$.

By (2.65), this implies principality.

Case: $e =$ `fix` $f:t$ `=>` `fn` $x:t_1$ `=>` e' The code for this case is

```
| infer VR (e as (fix f:t => fn x:t1 => e')) =
    let fun loop r =
            let val next? = infer (VR[f := r]) (fn x:t1 => e')
            in
              if subtypep next? r t then r
              else if next? = ns then ns
              else loop next?
            end
        val t1' -> t2 = t
    in
       if t1' ≠ t1 then ns
       else if ML type inference does not give rtom(VR) ⊢ e :: t1 -> t2
       then ns
       else loop (botfn (t1 -> t2))
    end
```

We will abbreviate `fn` $x:t_1$ `=>` e' as e''.

Suppose

$$\text{VR} \Vdash \texttt{fix}\ f{:}t\ \texttt{=>}\ e'' : k. \tag{2.68}$$

The last inference of this must be FIX-TYPE, so

$$t \text{ has the form } t_1 \to t_2 \tag{2.69}$$

and the premises of FIX-TYPE are

$$k \sqsubset t_1 \to t_2$$

and

$$\text{VR}[f := k] \vdash e'' : k. \tag{2.70}$$

We will show by induction on the execution of this case of `infer` the following properties of the argument r of `loop`:

$$r \leq k$$

and

$$\texttt{infer}\ (\text{VR}[f := r])\ e'' \text{ is not } \mathbf{ns}.$$

For the base case, $r = \texttt{botfn}\ t$. Soundness of `botfn` gives $r \leq k$, and Lemma 2.66 (Environment Modification) on page 81 applied to (2.70) gives $\text{VR}[f := r] \vdash e'' : k$. By Fact 2.35 (Splits of Arrows are Simple) on page 51, r has no useful splits; thus the outer induction hypothesis tells us that $\texttt{infer}\ (\text{VR}[f := r])\ e''$ is not $\mathbf{ns}$, which is what we wanted to show.

For the induction case, this call to `loop` is from the body of `loop`. Thus we can assume by induction that $r \leq k$ and we have to show that $next? \leq k$ and that $\texttt{infer}\ (\text{VR}[f := next?])\ e''$ is not $\mathbf{ns}$. Lemma 2.66 (Environment Modification) on page 81 starting with (2.70) gives

$$\text{VR}[f := r] \vdash e'' : k.$$

The outer induction hypothesis applies because Fact 2.35 (Splits of Arrows are Simple) on page 51 tells us r has no useful splits, so we have

$$\texttt{infer}\ (\text{VR}[f := r])\ e'' \text{ is not } \mathbf{ns}\text{; call it } next?$$

and

$$next? \leq k. \tag{2.71}$$

Lemma 2.66 (Environment Modification) on page 81 starting with (2.70) again gives

$$\text{VR}[f := next?] \vdash e'' : k$$

and the outer induction hypothesis (using Fact 2.35 (Splits of Arrows are Simple) on page 51 to conclude that *next?* has no useful splits) gives $\texttt{infer}\ (\text{VR}[f := next])\ e''$ is not $\mathbf{ns}$. This and (2.71) are our conclusions. This completes the inner induction.

Now we have everything we need to show that `infer` VR e is not ns. Theorem 2.54 (Inferred Types Refine) on page 68 applied to (2.68) gives a t' such that $k \sqsubset t'$ and

$$\text{rtom}(\text{VR}) \vdash e :: t'. \tag{2.72}$$

Lemma 2.10 (Unique ML Types) on page 31 gives $t = t'$. By (2.69) and (2.72), the `if` statements before the initial call to `loop` do not cause `infer` to return ns. By the most recent induction, the call to `loop` does not return ns. Thus

$$\texttt{infer VR } e \text{ is not ns.}$$

The most recent induction also gives

$$(\texttt{infer VR } e) \leq k.$$

Summarizing this subcase so far,

$$\begin{array}{l}\text{VR} \Vdash e : k \text{ implies} \\ \quad (\texttt{infer VR } e) \preceq k.\end{array}$$

By (2.65), this implies principality. □

The next theorem shows that `infer` always terminates. The case of this theorem dealing with `fix` statements uses Theorem 2.101 (Infer Returns Principal Type) on page 151.

Theorem 2.102 (Infer Terminates) *If*

all splits of types in VR *are useless*

then

`infer` VR e *always terminates.*

Proof: By induction on e. The cases are all very simple, except the case for `fix` statements.

Case: $e = y$ The code for this case is

```
fun infer VR y =
    if for some t we have VR(y) ⊏ t
    then VR(y)
    else ns
```

and termination is trivial.

Case: $e =$ `fn` x`:`t `=>` e' The code for this case is

```
| infer VR (fn x:t => e') =
    if there is a u such that rtom(VR)[x := t] ⊢ e' :: u
    then
       let val u = the unique u such that rtom(VR)[x := t] ⊢ e' :: u
           fun do_one r =
              sjoinf u {infer (VR[x := r']) e' | r' ∈ split r}
        in
           △fn {r → do_one r | r ∈ allrefs t and (do_one r) ≠ ns}
        end
    else ns
```

By soundness of `split`, all r' in `split` r have no useful splits. Thus, by induction hypothesis, the recursive calls to `infer` all terminate. Since principal splits are computable, all calls to `split` terminate. Since the refinements of an ML type are enumerable, calls to `allrefs` terminate. Thus this case of `infer` terminates.

Case: $e = e_1\ e_2$ The code for this case is

```
| infer VR (e1 e2) =
    let val r? = infer VR e1
        val k? = infer VR e2
    in
        if r? = ns or k? = ns
        then ns
        else
           let
              val u → t = rtom(r?)
              val u' = rtom(k?)
           in
              if u = u'
              then ifn r? k? u
              else ns
           end
    end
```

By induction hypothesis, the recursive calls to `infer` terminate. By soundness of `ifn` it always terminates. Thus this case of `infer` terminates.

Case: $e = c\ e'$ The code for this case is

```
| infer VR (c e') =
    let val k? = infer VR e'
        val t = the unique t such that c ::(def) t ↪ tc
    in
        △fn {rc | r ∈ allrefs t and subtypep k? r t and c :(def) r ↪ rc}
    end
```

All loops in this case of `infer` loop over finite sets, and our induction hypothesis tells us that the recursive call to `infer` terminates.

Case: $e =$ `case` e_0 `of` c_1 `=>` e_1 `|` ... `|` c_n `=>` e_n `end:`t The code for this case is

```
| infer VR (e as (case e0 of c1 => e1 | ... | cn => en end:t)) =
    if not rtom(VR) ⊢ e :: t then ns
    else let val r? = infer VR e0
         in if r? = ns then ns
            else let val rc = rconsimp (r?)
                 in
                     sjoinf t {ifn (infer VR eh) p u |
                                   h ∈ 1..n
                               and ch ::(def) u ↪ uc
                               and p ∈ allrefs u
                               and ch :(def) p ↪ rc}
                 end
         end
```

By induction hypothesis, all recursive calls to `infer` terminate. All other operations in this case are calls to functions that terminate or iterations over finite sets, so this case of `infer` terminates.

Case: $e = (e_1, \ldots, e_n)$ The code for this case is

```
| infer VR (e1, ..., en) =
    if for any i in 1..n we have infer VR ei = ns
    then ns
    else infer VR e1 * ... * infer VR en
```

By induction hypothesis, all recursive calls to `infer` terminate, so this case of `infer` terminates.

Case: $e =$ `elt_`m`_`n e' The code for this case is

```
  | infer VR (elt_m_n e') =
      let val k? = infer VR e'
      in
         if k? = ns then ns
         else let val k1 * ... * kn = tuplesimp (k?)
              in km end
      end
```

By induction hypothesis, the recursive call to `infer` terminates. Calls to `tuplesimp` always terminate. Thus this case of `infer` terminates.

Case: $e =$ `fix` $f{:}t$ `=>` `fn` $x{:}t_1$ `=>` e' The code for this case is

```
| infer VR (e as (fix f:t => fn x:t1 => e')) =
    let fun loop r =
            let val next? = infer (VR[f := r]) (fn x:t1 => e')
            in
              if subtypep next? r t then r
              else if next? = ns then ns
              else loop next?
            end
        val t1' -> t2 = t
    in
       if t1' <> t1 then ns
       else if ML type inference does not give rtom(VR) |- e :: t1 -> t2
       then ns
       else loop (botfn (t1 -> t2))
    end
```

We will abbreviate `fn` $x{:}t_1$ `=>` e' as e''.

If we never get to the call to `loop` in this case of `infer`, then we obviously terminate and return ns. Otherwise, Theorem 2.100 (Infer Returns Some Type) on page 145 and Lemma 2.99 (Fix Case of Infer is Well-Behaved) on page 144 tell us that the argument r to `loop` always refines $t_1 \to t_2$. By Fact 2.35 (Splits of Arrows are Simple) on page 51, r has no useful splits, so our induction hypothesis applies and tells us all recursive calls of the form `infer` $(\text{VR}[f := r])$ e'' terminate. Thus the computation progresses from each recursive call to `loop` to the next. Now we have to show that there are only finitely many recursive calls, and then we will know that the outer call to `loop` terminates.

Let $r_1, r_2, \ldots$ be the values of r in the successive recursive calls to `loop`. Thus $r_1 =$ `botfn` $(t_1 \to t_2)$. We will show by induction that for all $h > 0$ we have $r_h \leq r_{h+1}$.

The base case is trivial. Since $r_1 =$ `botfn` $(t_1 \to t_2)$, we know that r_1 is a subtype of any refinement of $t_1 \to t_2$. Earlier argument tells us that $r_2 \sqsubset t_1 \to t_2$, so this implies $r_1 \leq r_2$.

For the induction case, we can assume that $r_{h-1} \leq r_h$. By definition of `loop`, $r_{h+1} =$ `infer` $(\text{VR}[f := r_h])\ e''$. Theorem 2.100 (Infer Returns Some Type) on page 145 therefore gives

$$\text{VR}[f := r_h] \vdash e'' : r_{h+1}.$$

Then we can use Lemma 2.66 (Environment Modification) on page 81 and $r_{h-1} \leq r_h$ to get

$$\text{VR}[f := r_{h-1}] \vdash e'' : r_{h+1}.$$

Since $r_{h-1} \sqsubset t_1 \to t_2$, Fact 2.35 (Splits of Arrows are Simple) on page 51 tells us that r_{h-1} has no useful splits. Thus Theorem 2.101 (Infer Returns Principal Type) on page 151 gives

$$(\texttt{infer}\ (\text{VR}[f := r_{h-1}])\ e'') \preceq r_{h+1}.$$

By definition of `loop`, this is equivalent to

$$r_h \leq r_{h+1},$$

which is what we wanted to show. This completes the inner induction.

Repeated use of TRANS-SUB with the inner induction gives

$$h \leq j \text{ implies } r_h \leq r_j.$$

By definition of `loop` and soundness of `subtypep`, we would not get to iteration $h + 1$ if $r_{h+1} \leq r_h$; thus, for all h we have

$$r_{h+1} \not\leq r_h.$$

From this we can use the following reasoning to show that no two of the r_h's are equivalent. Suppose by way of contradiction that $r_h \equiv r_j$ where $h < j$. Then $h + 1 \leq j$, so we have $r_{h+1} \leq r_j$. Then we can use TRANS-SUB on this and $r_h \equiv r_j$ to get $r_{h+1} \leq r_h$. This contradicts our result from the previous paragraph, so we cannot have $r_h \equiv r_j$ when $h < j$.

By Theorem 2.90 (Finite Refinements) on page 115, the sequence $r_1, r_2, \ldots$ only contains representatives from finitely many equivalence classes of refinements of $t_1 \to t_2$. Since they are all from distinct equivalence classes, there must be only finitely many of them. Thus there are only finitely many r_h's, and `loop` and this case of `infer` always terminate. □

Chapter 3

Declaring Refinements of Recursive Data Types

3.1 Introduction

The previous chapter defined refinement type inference in terms of sets of refinement type constructors refining each ML type constructor. This chapter describes `rectype` declarations, which are a compact way to specify these sets of refinement type constructors and the operations on them.

We shall call the types appearing in `rectype` declarations *recursive types*. These types bear some resemblance to the recursive types of [AC90]; we compare the two systems on page 169.

Because refinement type constructors must be closed under intersection, we must either require `rectype` declarations to include enough definitions to ensure closure, or we need to allow the theory to introduce refinement type constructors that do not appear in any `rectype` statement. For example, the declaration

```
datatype bool = true () | false ()
rectype tt = true (runit)
    and ff = false (runit)
```

does not define a refinement type constructor for the intersection of tt and ff. It would better to automatically synthesize such a definition than to require the programmer to augment the `rectype` declaration. The theory below does this; the synthesized type is tt & ff. Here & is an infix operator that combines one or more of the identifiers defined by the `rectype` statement (which we shall call *recursive type constructors*) to form a refinement type constructor. Since & is infix, the assumption we have made so far that the `rectype` statement above defines the refinement type constructors tt and ff is true with this interpretation.

Another concern is interacting smoothly with the global environment used in the previous chapter. In that chapter, we wrote assertions like

$$\texttt{nil} \stackrel{\text{def}}{::} runit \hookrightarrow blist$$

as though they were simply true, instead of treating them as assumptions from an explicit list of assumptions, or environment; to put it another way, the environment was an implicit global variable. The environment never changed, so this was very convenient. In contrast, the declarations introduced in this chapter specify new assumptions to add to the environment. We could clarify the manipulation of the environment by making the environment explicit, but that would create notational problems when we refer to results from the previous chapter, so instead we will continue to manipulate it implicitly. Since we are describing changes to the environment, we have "old" assertions that are already in the environment and in this chapter we describe the "new" ones that will be added. The words "old" and "new" will be used consistently in this sense throughout this chapter.

With this distinction in mind, we can give the following grammar for `rectype` statements. The metavariable names in this chapter are slightly awkward because both "recursive" and "refinement" start with "r". We resolve the ambiguity by using "n" (standing for "new") in the names of metavariables concerned with recursive types. For example, in the grammar below, we use the terminal nrc to stand for recursive type constructors. We also use rc to stand for old refinement type constructors, r to stand for old refinement types, tc to stand for ML type constructors, and c to stand for new value constructors:

$$\begin{aligned} rstmt &::= \texttt{rectype } \textit{defn } \texttt{and ... and } \textit{defn} \\ defn &::= nrc \texttt{ = } enr \\ enr &::= rc \mid nrc \mid c\ \ (enr) \mid r \rightarrow enr \mid \\ &\quad\ \ enr * \ldots * enr \mid runit \mid \texttt{bottom } tc \mid \\ &\quad\ \ enr \ \&\ enr \mid enr \texttt{ | } enr \end{aligned}$$

In this grammar the name of the nonterminal enr stands for *extended recursive types* (which are slightly more flexible than the recursive types that will be introduced below). Notice that the metasymbol "|" is used in the grammar to define a language construct containing the character "`|`". We shall assume throughout that the syntactic operators & and `|` are associative, commutative, and idempotent; thus, for example, tt & ff and ff & tt are the same syntactic object.

The intuitive meaning of these declarations is fairly simple: think of them as a notation for defining sets of values. The recursive type constructor on the left hand side of the "=" is defined by the extended recursive type on the right hand side. The extended recursive type $c\ \ (enr)$ contains all values that can be constructed by applying the constructor c to some value in enr. The extended recursive type `bottom` (tc) contains no values and it refines tc; it should be distinguished from the identifier $\bot_{tc}$, which is the typeset form of the identifier $bottom\_tc$ and can in principle be given an arbitrary definition by the programmer (although defining it as anything other than `bottom` (tc) would be an unnecessary surprise). The extended recursive type enr_1 `|` enr_2 contains all values appearing in either enr_1 or

enr_2. The meanings of the other extended recursive types should be clear by analogy with refinement types.

The notation $c\ \ (enr)$ is meant to resemble the use of value constructors to construct values. We increase the resemblance by allowing $c\ \ (enr_1, \ldots, enr_n)$ as syntactic sugar for $c\ \ (enr_1 * \ldots * enr_n)$. To make parsing easier, we require the parentheses to always be present; this makes it possible to parse `rectype` statements without knowing in advance which identifiers are value constructors and which are refinement or recursive type constructors.

As we did for refinement types, we shall use $runit$ to stand for the empty tuple of recursive types. Comparing this grammar with the one for refinement types on page 30 makes it clear that all refinement types look like extended recursive types; although there is a natural correspondence between the two, it is best to think of them as distinct. Context will make it clear which is meant. An alternative would be to add notation to make the two kinds of types appear distinct; this seems too laborious.

The notation `bottom` tc gives a way to write types that contain no values; we shall say that these types are *empty*. If there are two or more value constructors, we can also write an empty type as an intersection; for example, given the datatype declaration

$$\texttt{datatype}\ blist\ \texttt{=}\ \texttt{cons of}\ bool * blist\ \ \texttt{|}\ \texttt{nil of}\ tunit$$

we can write the empty type as

$$\texttt{rectype}\ \bot_{blist}\ \texttt{=}\ \texttt{cons}\ (\top_{bool}, \bot_{blist})\ \&\ \texttt{nil}\ (runit)$$

However, it seems better to provide the `bottom` notation as well, since this is more direct approach. When we transform the syntax described here into a normal form, only the direct approach will be available.

In this thesis, we will require the declaration of a datatype and the unique `rectype` declaration specifying refinements of that datatype to appear together. A more general approach would allow declaring a datatype followed by some expressions using that datatype followed by a declaration of refinements of that datatype, or even `rectype` declarations that have their scope limited by a `let` statement. In the general case, two problems arise: what to do with the types of variables in the environment when entering the scope of a `rectype` declaration and what to do when we leave the scope of a `rectype` declaration. These problems seem solvable, but nevertheless beyond the scope of this thesis. Because we forbid separating corresponding `datatype` and `rectype` declarations, when we analyze a `rectype` declaration it is possible to make a clear distinction between "new" value constructors and ML type constructors and "old" ones: the new constructors appear in the associated `datatype` statement, and the old ones do not.

The syntax of `rectype` statements above outlaws recursion on the left hand side of $\rightarrow$ by only permitting old refinement types on the left hand side of $\rightarrow$. This avoids situations

where there is no obvious fixed point of a declaration; for example, suppose the restriction were lifted and consider the declaration

$$\texttt{datatype } d \texttt{ = A of } d \to bool \texttt{ | B of } tunit$$
$$\texttt{rectype } r \texttt{ = A of } r \to tt$$

From an intuitive point of view, it is entirely unclear how to determine whether a particular value is in r because as we include more values in r, the definition of r tells us there are fewer values in r. Formally, the problem with recursion on the left hand side of $\to$ is that the definition of membership of a value in a recursive type in Figure 3.2 ceases to be a monotone function, so we no longer know that it has a fixed point. This is discussed in more detail on page 182 after we present that definition.

3.1.1 Outline of this Chapter

A `rectype` declaration is accepted by type inference if it satisfies some minor semantic restrictions described in the next section and it can be rewritten as a set of definitions of the form:

$$\begin{aligned} defn &::= nrc \succeq c(nr) \mid \texttt{bottom } (tc) \\ nr &::= r \to nr \mid nr \;\&\; nr \mid rc \mid nrc \mid nr * \ldots * nr \mid runit \end{aligned}$$

Unlike the previous grammar, this one only allows value constructors at the top level, and it disallows the symbol "|". We have also replaced the "=" with "$\succeq$"; this is meant to imply that we now allow multiple declarations of a type for a given nrc to (roughly speaking) mean that nrc stands for the union of all the definitions that appear. The meaning is formally defined in Section 3.2. For example, rewriting the declaration

$$\texttt{datatype } blist \texttt{ = cons of } bool * blist \texttt{ | nil of } tunit$$
$$\texttt{rectype } bev \texttt{ = cons } (\top_{bool} * \texttt{cons } (\top_{bool} * bev)) \texttt{ | nil } (\textit{runit})$$

starts by creating a new type name (the implementation will select names that look like $g398$) and results in the set

$$\begin{aligned} \{&bev \succeq \texttt{cons}(\top_{bool} * g398), \\ &bev \succeq \texttt{nil}(runit), \\ &g398 \succeq \texttt{cons}(\top_{bool} * bev), \\ &\top_{blist} \succeq \texttt{cons}(\top_{bool} * \top_{blist}), \\ &\top_{blist} \succeq \texttt{nil}(runit)\} \end{aligned}$$

In Section 3.3, we describe how to infer that some recursive types are empty. For example, we can infer that in the presence of the declaration above, the recursive type bev & $g398$ contains no values. This inference system is only valid because our constructors are eager; if they were lazy, the type bev & $g398$ would contain the infinite value

`cons (true (), cons (true (), ...))`, which could be constructed by using a fixed point operator.

In Section 3.4, we describe how to infer when one recursive type is a subtype of another. This inference system can reason about empty types; for example, if we add the definition

$$bem \succeq \texttt{nil}(runit)$$

we can infer $bev \ \& \ g398 \leq bem$.

In Section 3.5, we describe how to infer that one type is contained in a union of other smaller types. For example, we can infer $\top_{blist} \asymp \{bev, g398\}$.

In Section 3.6, we define the new refinement type constructors in terms of the recursive type constructors appearing in the declaration, and we prove that all of the assumptions made in Chapter 2 about the behavior of refinement type constructors hold when they are defined by `rectype` statements. We also prove that whenever a value has a refinement type, it has the corresponding recursive type.

3.1.2 Related Work

We can think of a recursive type as a recognizer for a sublanguage of the language of values; in this sense, a recursive type is similar to a regular tree automaton as described in [GS84]. One difference is that our language of values includes functions; another difference is that our procedure for deciding subtypes for recursive types is weaker than the decision procedures for deciding inclusion of regular tree automata in [GS84], even for recursive types that contain no function types. See the example on page 193.

The algorithms presented in this Chapter are similar to the ones in [TZ91]. Our recursive types differ from their term grammars in that we have function types but they do not, and their term grammars are closed under union and complement but our recursive types are not. Some of the proofs below use an induction principle that appears in their paper, specifically induction on the pair (complement of the trail, some tree) ordered lexicographically.

The algorithms presented in this Chapter also resemble the ones in [AC90]. The abstract declarations appearing here are very similar to the regular systems described in that paper, and our algorithm for subtyping recursive types is a version of the algorithm described on page 24 of that paper, modified to deal with the features we have added to our type system. Our recursive types disallow the recursion on the left hand side of arrows that is allowed in [AC90], and our system has intersections, which do not appear in [AC90]. The proof in this chapter that the algorithm used here is correct does not resemble theirs at all; they reason about finite approximations to infinite trees to show that their algorithm is consistent with another axiomatization of subtyping, whereas we have axioms for determining when a value has a recursive type, and we prove that the inclusion relation from this algorithm is consistent with membership of values in types.

In both [AC90] and this chapter, there are two conceptually distinct fixed points in the definition of membership of a value in a recursive type. One fixed point converts the recursive type to a potentially infinite non-recursive type; in [AC90], this is a least fixed point. In this Chapter, we require each recursive definition to have the form $nrc \succeq c(\ldots)$; since the constructor c is always present, the recursion makes progress and the infinite tree is uniquely determined. (In [AC90], the recursive type is rewritten to make the fixed point unique before the subtyping algorithm is used.)

The second fixed point determines whether a value is in this potentially infinite non-recursive type. In this chapter, the fixed point is a greatest fixed point. In [AC90], enough explicit types appear in the terms to uniquely determine the fixed point. Using a greatest fixed point is appropriate, since we want to assign as many types to as many terms as possible while preserving soundness.

There are four important relations axiomatized in this Chapter: membership of a value in a recursive type, emptyness of a recursive type, subtyping for recursive types, and splitting for recursive types. All of these are greatest fixed points of the axiom system, rather than the customary least fixed points. Informally, this means infinite proof trees are permitted. Formally, we think of each inference system as a monotone function and consider a judgement to be valid if it is a member of the greatest fixed point of the function. A proof technique commonly used in the literature (and in this thesis) with greatest fixed points is co-induction, as described in [MT91a], among other places.

3.2 Abstract Declarations

In the previous chapter, we assumed that information about the primitives $\overset{\text{def}}{:}$, $\overset{\text{def}}{\leq}$, and so forth appeared in a global environment. When type inference encounters `datatype` and `rectype` statements, the global environment must be updated appropriately. In this chapter, we will assume that the `datatype` statement has already been added to the environment, and we will describe how to add the `rectype` statement.

The proofs and inference systems are simpler if we simply assume that & for recursive types is commutative, associative, and idempotent.

Declarations given by the programmer need to be manipulated in several ways before they become regular enough for simple algorithms to apply to them. In this section we will informally describe how the user's declarations are converted to a normal form called *abstract declarations*, and we will define well-formedness for abstract declarations. All of the algorithms described in future sections take well-formed abstract declarations as input.

3.2.1 Expansion

Circular definitions of recursive type constructors are potentially confusing. For example, consider the declaration

$$\begin{array}{l}\texttt{datatype } d \texttt{ = D of } d \\ \qquad \texttt{and } e \texttt{ = E of } e \\ \texttt{rectype } loop \texttt{ = } loop\end{array}$$

We could decide that the fixed point by which we give meaning to these declarations is a least fixed point, as was done in [AC90]; with this interpretation this declaration would mean that $loop$ is an empty type. Alternatively, we could decide that it is a greatest fixed point, in which case $loop$ should contain all of the refinements of some ML type. However, there is no natural way to determine which ML type $loop$ refines, so instead this declaration is considered an error. In this Subsection we will detect all errors of this kind by making sure each definition of a recursive type constructor makes progress before recurring.

Define the *toplevel* of an extended recursive type to be the outermost subterms of the recursive type that do not use the "&" or "|" operators. For example, the toplevel of the extended recursive type

$$\texttt{cons } (\top_{bool} * bem) \,\&\, (\texttt{cons } (tt * bnem) \mid bem)$$

consists of the subterms $\texttt{cons } (\top_{bool} * bem)$, $\texttt{cons } (tt * bnem)$, and bem. To perform expansion, repeatedly replace all refinement type constructors at the toplevel with their definitions until there are no refinement type constructors at the toplevel, or some definition is expanded more than once. If a definition is expanded more than once, we have a circular definition and the `rectype` declaration is rejected as meaningless.

For example, the declaration

$$\begin{array}{l}\texttt{datatype } nat \texttt{ = Succ of } nat \texttt{ | Zero of } tunit \\ \texttt{rectype } loop1 \texttt{ = } loop2 \\ \qquad \texttt{and } loop2 \texttt{ = } loop1\end{array}$$

is rejected because attempts to expand the definitions of both $loop1$ and $loop2$ fail to terminate. On the other hand, the declaration

$$\begin{array}{l}\texttt{datatype } nat \texttt{ = Succ of } nat \texttt{ | Zero of } tunit \\ \texttt{rectype } loop1 \texttt{ = } loop2 \\ \qquad \texttt{and } loop2 \texttt{ = Succ } (loop1)\end{array}$$

is accepted and the result of this manipulation is

$$\begin{array}{l}\texttt{datatype } nat \texttt{ = Succ of } nat \texttt{ | Zero of } tunit \\ \texttt{rectype } loop1 \texttt{ = Succ } (loop1) \\ \qquad \texttt{and } loop2 \texttt{ = Succ } (loop1).\end{array}$$

3.2.2 Flattening

Declarations given by the programmer will often require inventing new refinement type names to get the expected effect. For example, with the declaration

$$\texttt{datatype } \mathit{blist} \texttt{ = cons of } \mathit{bool} * \mathit{blist} \texttt{ | nil of } \mathit{tunit}$$
$$\texttt{rectype } \mathit{bev} \texttt{ = cons } (\top_{\mathit{bool}} * \texttt{cons } (\top_{\mathit{bool}} * \mathit{bev})) \texttt{ | nil } (\textit{runit})$$

we would expect the expression `cons (true (), cons (true (), nil ()))` to have the refinement type bev. If the only refinement of $blist$ is bev, we cannot infer this type for this expression because we have no type for the expression `cons (true (), nil ())`. To get the expected type for `cons (true (), cons (true (), nil ()))`, we need to automatically add another refinement of $blist$. In practice, the new refinement would be given a nonmnemonic name like $g398$, and the `rectype` declaration would be treated as though it were written

$$\texttt{datatype } \mathit{blist} \texttt{ = cons of } \mathit{bool} * \mathit{blist} \texttt{ | nil of } \mathit{tunit}$$
$$\texttt{rectype } \mathit{bev} \texttt{ = cons } (\top_{\mathit{bool}} * g398) \texttt{ | nil } (\textit{runit})$$
$$\texttt{and } g398 \texttt{ = cons } (\top_{\mathit{bool}} * \mathit{bev})$$

(The programmer can define a usable name for odd length lists by doing this expansion by hand, using some mnemonic name such as "bod" in place of "$g398$".) The manipulation of the `rectype` statement that adds these new recursive type constructors is called *flattening*.

The problem is that we have value constructors that are not at the toplevel, and the solution is to introduce new recursive type constructors until all value constructors are at the top level. To describe this formally, we will have to speak in terms of a context C, which is an extended recursive type with a hole. For example, we can write

$$\texttt{cons } (\top_{\mathit{bool}} * \texttt{cons } (\top_{\mathit{bool}} * \mathit{bev})) \texttt{ | nil } (\textit{runit})$$

as $C[\top_{bool}]$, where $C[\cdot] = \texttt{cons } (\cdot * \texttt{cons } (\top_{\mathit{bool}} * \mathit{bev})) \texttt{ | nil } (\textit{runit})$. With this definition, we can formally specify how to flatten a `rectype` declaration: whenever we encounter a definition of the form

$$nrc \texttt{ = } C[nr]$$

where the $\cdot$ in $C[\cdot]$ does not appear at the toplevel, but all toplevel subexpressions of nr have the form $c\ (nk)$ or `bottom` (tc), we choose a new recursive type constructor nkc and replace this definition with the two definitions

$$nrc \texttt{ = } C[nkc]$$
$$nkc \texttt{ = } nr$$

The requirement that $\cdot$ does not appear at the toplevel of $C[\cdot]$ is necessary ensure that the manipulation actually makes the `rectype` simpler; without the requirement, the result of applying this manipulation to the above example could be

$$\begin{array}{l}\texttt{datatype } blist \texttt{ = cons of } bool * blist \texttt{ | nil of } tunit \\ \texttt{rectype } bev \texttt{ = } g398 \\ \quad \texttt{and } g398 \texttt{ = cons } (\top_{bool} * \texttt{cons } (\top_{bool} * bev)) \texttt{ | nil } (\textit{runit})\end{array}$$

which is hardly an improvement.

The requirement that all toplevel subexpressions of nr have the form $c\ (nk)$ or `bottom` (tc) is necessary to ensure that all refinement type constructors created by this manipulation refine an ML type constructor, instead of refining some ML type. For example, without this restriction the result of applying this manipulation to the above example could be

$$\begin{array}{l}\texttt{datatype } blist \texttt{ = cons of } bool * blist \texttt{ | nil of } tunit \\ \texttt{rectype } bev \texttt{ = cons } (\top_{bool} * \texttt{cons } (g398)) \texttt{ | nil } (\textit{runit}) \\ \quad \texttt{and } g398 \texttt{ = } \top_{bool} * bev\end{array}$$

which will not satisfy the semantic restrictions that appear below because the new recursive type constructor $g398$ refines $bool * blist$, which is not a new ML type constructor.

3.2.3 Simplification

Now we can manipulate the `rectype` statement to ensure that the toplevel of each definition is simply a call to a value constructor or `bottom`, rather than an intersection or union of calls to value constructors or `bottom`. We can also eliminate some unions; any unions not eliminated by this manipulation cause an error.

Repeat the following rewrites until none of them apply:

- If the definition has the form

  $$nrc \texttt{ = } enr_1 \texttt{ | } enr_2$$

  replace it with the two definitions

  $$\begin{array}{l}nrc \texttt{ = } enr_1 \\ nrc \texttt{ = } enr_2 .\end{array}$$

- If the definition has the form

  $$nrc \texttt{ = } c\ (enr_1) \,\&\, c\ (enr_2) \,\&\, \ldots$$

replace it with

$$nrc \texttt{ = } c\ \ (enr_1 \,\&\, enr_2) \,\&\, \ldots$$

- If the definition has the form

$$nrc \texttt{ = } c_1\ \ (enr_1) \,\&\, c_2\ \ (enr_2) \,\&\, \ldots$$

where c_1 and c_2 are different, replace it with

$$nrc \texttt{ = bottom } tc$$

where tc is the ML type of the result of c_1.

These rewrites will rewrite many `rectype` statements so each definition has the form nrc `=` $c\ \ (enr)$ or nrc `=` `bottom` (tc). If they do not, then the `rectype` statement is considered meaningless; for example, the results of rewriting may have the form nrc `=` $enr_1 \rightarrow enr_2$ or nrc `=` $c\ \ (enr_1)\ \&\ (enr_2 * enr_3)$. These are and should be disallowed; the former because the user has apparently attempted to declare a new refinement of $\rightarrow$, and the latter because the new recursive type constructor must refine both the output type of c (which must be a datatype) and some tuple type.

3.2.4 Adding Top

Suppose we declare the booleans as

```
datatype bool = true of tunit | false of tunit
rectype tt = true (runit)
    and ff = false (runit).
```

If this only gives rise to the refinement type constructors tt, ff, and tt & ff refining $bool$, then there would be no type for an expression when refinement type inference cannot infer that it always evaluates to `true ()` or it always evaluates to `false ()`. For example, most calls to the function `samelength` defined by

```
fun samelength (cons (x, tlx)) (cons (y, tly)) = samelength tlx tly
  | samelength nil nil = true
  | samelength _ _ = false
```

will get a type error. Our options at this point are to declare that most calls to `samelength` cause a type error unless the user adds a definition

$$\ldots \texttt{ and } \top_{bool} \texttt{ = true } (runit) \texttt{ | false } (runit),$$

or we could implicitly add the definition. Since there will often be expressions where refinement type inference does not deduce precise information, we choose to implicitly add definitions of catch-all types like $\top_{bool}$ to every `rectype` declaration.

When the value constructors have arguments, the added definitions will mention the maximal refinement of other refinement types. For example, the definition we would add to

$$\texttt{datatype } d \texttt{ = C of } bool$$
$$\texttt{rectype } dtrue \texttt{ = C } (tt)$$

would be

$$\ldots \texttt{ and } \top_d \texttt{ = C } (\top_{bool}).$$

When the datatype includes functions, the catch-all refinement type will have to have minimal refinement types on the left hand side of each arrow, as well. For example, the implicit definition of the catch-all type for the declaration

$$\texttt{datatype } d \texttt{ = C of } bool \rightarrow bool$$

is

$$\texttt{rectype } \top_d \texttt{ = C } ((tt \,\&\, ff) \rightarrow \top_{bool})$$

and not

$$\texttt{rectype } \top_d \texttt{ = C } (\top_{bool} \rightarrow \top_{bool})$$

because the latter does not assign a type to an expression `C x` when `x` has the type $tt \rightarrow tt$. As explained in Subsection 2.7.2 on page 74, we cannot yet construct values with the least type $tt \rightarrow tt$, but we will be able to in Chapter 6.

The general procedure for creating catch-all types is straightforward and will not be given here. It starts to break down when we introduce polymorphic type constructors; see Subsection 5.8.3 on page 272.

This procedure is not meaningful with a `datatype` declaration that is recursive on the left hand side of $\rightarrow$ such as

$$\texttt{datatype } d \texttt{ = A of } d \rightarrow bool \texttt{ | B of } tunit$$

because the generated `rectype` declaration would have recursion on the left hand side of the $\rightarrow$, which is not consistent with the grammar given above for `rectype` statements. The user cannot use a `rectype` statement to specify refinements of d either, for the same reason. The best approach seems to be to give datatypes like this exactly one refinement, which would be called $\top_d$ in this case, and to give trivial definitions of the primitives used in Chapter 2 that satisfy the assumptions made.

3.2.5 Definition of Abstract Declarations

This chapter deals with recursive types, which we define in terms of recursive type constructors and refinement types. We shall use the following metavariables in this chapter:

nrc, nkc, npc	Recursive type constructors.
$nrcs, nkcs, npcs$	Sets of recursive type constructors.
nr, nk, np, nq	Recursive types.
nrs, nks, nps	Sets of recursive types.

This naming scheme is meant to be mnemonic; "n" stands for "new", "s" stands for "set", "c" stands for "constructor", and "r", "k", "p" and "q" simply distinguish multiple names of each type. We will also occasionally use metavariables defined in the previous chapter.

After we rewrite `rectype` statement given by the programmer as described above, we can summarize the `rectype` statement as a set D of expressions of the form $nrc \succeq c(nr)$ or the form $nrc \succeq \text{bottom}(tc)$.

For example, the declaration

$$\begin{array}{l}\texttt{datatype } blist \texttt{ = cons of } bool * blist \texttt{ | nil of } \textit{tunit} \\ \texttt{rectype } bev \texttt{ = cons } (\top_{bool}\texttt{, cons } (\top_{bool}\texttt{, } bev)) \texttt{ | nil } (\textit{runit}) \\ \quad \texttt{and } bod \texttt{ = cons } (\top_{bool}\texttt{, } bev) \\ \quad \texttt{and } \bot_{blist} \texttt{ = bottom } (blist)\end{array}$$

corresponds to the abstract declaration

$$\begin{array}{l}\{\top_{blist} \succeq \texttt{cons}(\top_{bool} * \top_{blist}), \\ \ \top_{blist} \succeq \texttt{nil}(runit), \\ \ bev \succeq \texttt{cons}(\top_{bool} * bod), \\ \ bev \succeq \texttt{nil}(runit), \\ \ bod \succeq \texttt{cons}(\top_{bool} * bev), \\ \ \bot_{blist} \succeq \text{bottom}(blist)\}.\end{array}$$

The environments of most of the type inference rules below will include an abstract declaration, usually called D. There will be no corresponding description of the ML type environment; instead, we will assume that appropriate assumptions of the form $c \overset{\text{def}}{::} t \to tc$ reflect the `datatype` declaration when c is new. We do this because this thesis is not concerned with ML type inference and the extra notation does not seem worthwhile. A complete description of an ML dialect that included `rectype` declarations should have an explicit environment that includes descriptions of the `datatype` declarations in effect as well as the `rectype` declarations.

AND-RECREFINES: $$\frac{D \vdash nr \sqsubset t \qquad D \vdash nk \sqsubset t}{D \vdash nr \;\&\; nk \sqsubset t}$$

ARROW-RECREFINES: $$\frac{r \sqsubset t_1 \qquad D \vdash nr \sqsubset t_2}{D \vdash r \to nr \sqsubset t_1 \to t_2}$$

NEW-RECREFINES: $$\frac{\begin{array}{l}\text{for all } c \text{ and } nr \text{ such that } nrc \succeq c(nr) \in D \\ \quad \text{there is a } t \text{ such that } c \stackrel{\text{def}}{::} t \hookrightarrow tc \\ \quad \text{for all } tc' \text{ such that } nrc \succeq \text{bottom}(tc') \in D \\ \quad \text{we have } tc = tc'\end{array}}{D \vdash nrc \sqsubset tc}$$

OLD-RECREFINES: $$\frac{rc \stackrel{\text{def}}{\sqsubset} tc}{D \vdash rc \sqsubset tc}$$

TUPLE-RECREFINES: $$\frac{\text{for all } i \text{ we have } D \vdash nr_i \sqsubset t_i}{D \vdash nr_1 * \ldots * nr_n \sqsubset t_1 * \ldots * t_n}$$

Figure 3.1: Monomorphic Recursive Type Refinement Rules

3.2.6 Well-formedness

In this section, we will give some conditions that abstract declarations used in this chapter must satisfy. `Rectype` declarations giving rise to abstract declarations that do not satisfy these conditions are rejected by type inference.

Given an abstract declaration, we must first check that it is well-formed. Since this thesis is about refinement type inference and not ML type inference, we will assume without further ado that assertions of the form $c \stackrel{\text{def}}{::} t \hookrightarrow tc$ derived from the `datatype` statement are available. For instance, given the declaration

`datatype` *blist* `= cons of` *bool* $*$ *blist* `| nil of` ***tunit***

we should immediately have the assertions

$$\texttt{cons} \stackrel{\text{def}}{::} (bool * blist) \hookrightarrow blist$$
$$\texttt{nil} \stackrel{\text{def}}{::} tunit \hookrightarrow blist.$$

Given these assertions, it is possible to use the inference rules in Figure 3.1 to infer that certain recursive types refine certain ML types.

These rules are analogous to the monomorphic refinement rules in Figure 2.3 on page 31, except we have added a rule NEW-RECREFINES which is not similar to any rule from

Figure 2.3. This rule makes clear the purpose of the "bottom" declarations that can appear in D; they constrain the ML type of recursive type constructors that would otherwise be entirely absent from D. This is the only place we will use the "bottom" declarations. Without these declarations, a completely empty recursive type constructor would not appear at all in the abstract declaration, so it could refine all ML type constructors, which would make Fact 3.9 (Recursive Unique ML Types) on page 179 false.

As in Chapter 2, we will consider $runit$ and $tunit$ to be tuples of zero elements, so we can use the TUPLE-RECREFINES rule to infer $D \vdash runit \sqsubset tunit$.

Now that we can determine when a recursive type refines an ML type, we can say what it means for an abstract declaration to be well-formed. An abstract declaration is well-formed if the next six conditions all hold. These conditions can all be easily checked by a program.

First we require all definitions in the abstract declaration to be consistent with the ML types of the value constructors:

Condition 3.1 (Refinement Consistency) *If D is well-formed then for all nrc $\succeq c(nr) \in D$, there are must be t and tc such that $c \stackrel{\text{def}}{::} t \hookrightarrow$ tc and $D \vdash$ nrc $\sqsubset$ tc and $D \vdash$ nr $\sqsubset t$.*

The distinction between "new" and "old" constructors mentioned earlier is only useful if the new constructors are limited in how they interact with the old ones. An appropriate restriction is:

Condition 3.2 (New Recursive Type Constructors Defined) *Every well-formed abstract declaration must define all new recursive type constructors.*

We need this because there is no way to determine the ML type refined by a new recursive type constructor that does not appear in the declaration. This restriction is satisfied naturally if the set of new recursive type constructors is taken as the recursive type constructors that appear in the abstract declaration.

Condition 3.3 (New Value Constructors Defined) *Every well-formed abstract declaration must mention all new value constructors.*

Without this restriction, the behavior of the new value constructors on refinement types would not be determined. This restriction is enforced naturally when catch-all recursive types are added.

Condition 3.4 (New Value Constructors Only) *Every well-formed abstract declaration must not mention any old value constructors.*

Without this, the abstract declaration could define new refinements of old ML types. For example:

```
datatype bool = true of tunit | false of tunit
... some code ...
datatype blist = cons of bool * blist | nil of tunit
rectype tt = true (runit)
```

We have several more conditions that simply formalize some of the behavior of `datatype` declarations. This restriction prevents incrementally declaring refinements of existing data types:

Condition 3.5 (New Value Constructors Closed) *The output type of each new value constructor must be a new ML type constructor.*

Condition 3.6 (Declarations are Finite) *All well-formed abstract declarations are finite.*

The abstract declaration, as written, gives one or more definitions for some of recursive type constructors. It is also possible to think of it as giving one or more definitions for some intersections of recursive type constructors; we call the set of all of the intersections the *closure* of D, and formally define it as follows:

Definition 3.7 (Intersection Membership) *Define $\overline{D}$ to be the set with elements of the form nrc_1 & ... & $nrc_n \succeq c(nr_1$ & ... & $nr_n)$ where for i between 1 and n we have $nrc_i \succeq c(nr_i) \in D$.*

Simple reasoning tells us that Condition 3.1 (Refinement Consistency) on page 178 extends naturally to intersections of recursive type constructors:

Fact 3.8 (Intersection Refines) *If $c \stackrel{\text{def}}{::} t \hookrightarrow tc$ and $\&nrcs \succeq c(nr) \in \overline{D}$ then*

$$D \vdash nr \sqsubset t$$

and

$$D \vdash \ \&\, nrcs \sqsubset tc.$$

An analogue of Lemma 2.10 (Unique ML Types) on page 31 holds for recursive types:

Fact 3.9 (Recursive Unique ML Types) *If $D \vdash nr \sqsubset t$ and $D \vdash nr \sqsubset u$ then $t = u$.*

The proof of this is a straightforward induction on nr.

$$\text{AND-RECVALUE:}\quad \frac{nrs \text{ has two or more elements} \quad \text{for each } nr \text{ in } nrs \text{ we have } D \vdash v \in nr}{D \vdash v \in \&nrs}$$

$$\text{ABS-RECVALUE:}\quad \frac{\text{for all } v \text{ and all } v' \text{ we have } \vdash v : r \text{ and } (\texttt{fn}\ x{:}t\ \texttt{=>}\ e)\ v \Rightarrow v' \text{ imply } D \vdash v' \in nr}{D \vdash (\texttt{fn}\ x{:}t\ \texttt{=>}\ e) \in r \to nr}$$

$$\text{NEW-RC-RECVALUE:}\quad \frac{nrc \succeq c(nr) \in D \qquad D \vdash v \in nr}{D \vdash c\ v \in nrc}$$

$$\text{OLD-RC-RECVALUE:}\quad \frac{\cdot \vdash v : rc}{D \vdash v \in rc}$$

$$\text{TUPLE-RECVALUE:}\quad \frac{\text{for all } i \text{ we have } D \vdash v_i \in nr_i}{D \vdash (v_1, \ldots, v_n) \in nr_1 * \ldots * nr_n}$$

Figure 3.2: Whether a Value is in a Recursive Type; Greatest Fixed Point

3.2.7 Meaning of Recursive Types

We can think of recursive types as standing for sets of values. In this section we will specify when a value is in a recursive type. For technical reasons described below, the inference system must be given an unusual interpretation that permits infinite proof trees. The inference system is also somewhat unusual in that some inferences can have infinitely many premises. Fortunately, this inference system does not need to be decidable. First we will explain why we need infinitely tall inference trees, and how to formalize this. Then we will explore various alternatives to the rule with infinitely many premises.

If one attempts to write inference rules for proving that a value has a recursive type, apparent success comes quickly. If we write "with the abstract declaration D, the value v has the recursive type nr" as $D \vdash v \in nr$, then we get the inference rules in Figure 3.2. (The need for the requirement of two or more elements in nks in the AND-RECVALUE rule and the meaning of the phrase "Greatest Fixed Point" in the caption will be explained in a moment.)

An ordinary interpretation of these inference rules works well for all values without functions embedded in them. Unfortunately, it is possible to use function objects to define possibly infinite lazy lists in ML, and a straightforward interpretation of the inference rules in Figure 3.2 draws wrong conclusions in this case. We can declare possibly infinite lazy lists of booleans with the declaration

$$\texttt{datatype}\ \textit{lazy}\ \texttt{=}\ \texttt{A of}\ \ tunit \to (bool * lazy)$$

and we can distinguish lazy lists where all elements are `true ()` with this declaration:

$$\texttt{rectype } \textit{alltrue} \texttt{ = A } (runit \rightarrow tt * alltrue)$$

The corresponding abstract declaration is

$$D = \{alltrue \succeq \texttt{A}(runit \rightarrow tt * alltrue)\}.$$

If the function object in a value with ML type *lazy* fails to terminate, then it vacuously satisfies the ABS-RECVALUE rule. Thus if we let

$$v_0 = \texttt{A (fn x => (fix f => fn x => (f x)))}$$

we have

$$D \vdash v_0 \in alltrue$$

and if we let $v_{i+1} = \texttt{A (fn x => (true (), } v_i\texttt{))}$ for $i > 0$, we also have

$$D \vdash v_i \in alltrue.$$

However, because the normal interpretation of inference systems disallows infinite proofs, we cannot use the normal interpretation of this system to give a type to the infinite lazy list

$$\texttt{A (fn \_ => ((fix f => fn \_ => (true (), A f)) ())).}$$

There are several possible ways to deal with this. In theory, one could imagine a type system that gives no type to the infinite lazy list. Distinguishing the infinite lazy list from lists that have a finite number of elements followed by an infinite loop when the next one is fetched is equivalent to the halting problem, so that type system would also have to give no type to some finite lazy lists. This seems awkward.

Instead, we use an informal interpretation of the above inference system that permits infinite proofs. Normally, the relation defined by an inference system is considered to be the least relation consistent with the inference rules. Instead, we will interpret it as the greatest relation consistent with the inference rules. This is the cause of the restriction of AND-RECVALUE to sets of two or more elements; if we allow sets of one element, then the conclusion of the rule is the same as the premise, and the greatest fixed point would include all possible conclusions because for any value v and any recursive type r we would have the infinite inference tree

$$\dfrac{\dfrac{\dfrac{\cdots}{D \vdash v \in r}\,[\text{AND-RECVALUE}]}{D \vdash v \in r}\,[\text{AND-RECVALUE}]}{}$$

Formally, we interpret this inference system as the greatest fixed point of a function. Take D as fixed for the time being. A greatest fixed point must be within some universe; let our universe U be the set of all possible pairs of the form (v, nr). We can encode the inference system in Figure 3.2 as a function F mapping subsets of U to subsets of U. If we abbreviate "nr has the form e" as "$nr \propto e$", the part of the definition of F corresponding to the AND-RECVALUE and ABS-RECVALUE rules is:

$(v, nr) \in F(Q)$ if and only if
$(nr \propto \&nks$ where nks has 2 or mere elements, and
for all nk in nks we have $(v, nk) \in Q)$
or
$(nr \propto r \to nk$ and $v \propto$ `fn` $x : t$ `=>` `e` and
for all v'' and v' we have
if $\cdot \vdash v'' : r$
and $(\texttt{fn}\ x : t\ \texttt{=>}\ e)\ v'' \Rightarrow v'$
then $(v', nk) \in Q)$
or
... omitted cases ...

where the omitted cases are always false if $nr \propto \&nks$ where nks has 2 or more elements, or $nr \propto r \to nk$. With this definition of F, we say $D \vdash v \in r$ if (v, r) is in the greatest fixed point of F, which we shall write as gfp(F).

It is easy to see that F is monotone. As we admit more premises of the form $D \vdash v \in r$, we can use the inference rules in Figure 3.2 to infer more conclusions of that form. By contrast, if we allow recursion on the left hand side of arrows, the natural version of ABS-RECVALUE would be

$$\frac{\text{for all } v \text{ and all } v' \text{ we have } D \vdash v \in nk \text{ and } (\texttt{fn}\ x : t\ \texttt{=>}\ e)\ v \Rightarrow v' \text{ imply } D \vdash v' \in nr}{D \vdash (\texttt{fn}\ x : t\ \texttt{=>}\ e) \in nk \to nr}$$

which is not monotone, since we have the premise $D \vdash v \in nk$ on the left hand side of an implication.

Another option that seems attractive at first is defining membership of a function in a recursive type in terms of some other type inference system. More specifically, we would say that `fn` $x : t$ `=>` e has the type $r \to nr$ if, in some sense, when we assume that x has the type r we can infer that e has the type nr. Unfortunately, we do not have a type inference system on hand that infers when an expression with free variables has a recursive type. We could make a recursive type inference system analogous to the refinement type inference system in Chapter 2, but the description of such a system might be about as large as Chapter 2. This inference system, on the other hand, is concise and sufficient for our purposes.

We use co-induction to reason about these greatest fixed points, as described in [MT91b, page 216]:

Fact 3.10 (Co-induction) *Let U be any set, and let F be a monotonic function mapping subsets of U to subsets of U. For any $Q \subset U$, in order to prove $Q \subset$ gfp(F), it is sufficient to prove $Q \subset F(Q)$.*

The first co-induction in this chapter is Theorem 3.20 (Emptyness Consistency II) on page 190.

Before we turn away from membership of values in recursive types, we note that co-induction is not necessary in the simple proof of an extended version of NEW-RC-RECVALUE that applies to intersections of recursive type constructors:

Fact 3.11 (Intersection Value Membership) *If &nrcs* $\succeq c(nr) \in \overline{D}$ *and* $D \vdash v \in nr$ *then* $D \vdash c\ v \in$ *&nrcs.*

3.3 Empty Types

The value constructors in Standard ML are eager, but the simplest type systems are more appropriate for lazy value constructors. The algorithm introduced in this section allows `rectype` statements to ignore certain distinctions that are unimportant in SML, but would be important if we had lazy value constructors. For example, if we have the declarations

$$\begin{array}{l}\texttt{datatype } blist \texttt{ = cons of } bool * blist \texttt{ | nil of } runit \\ \texttt{rectype } bev \texttt{ = cons } (\top_{bool} * bod) \texttt{ | nil } (\textit{runit}) \\ \quad \texttt{and } bod \texttt{ = cons } (\top_{bool} * bev) \\ \quad \texttt{and } bnem \texttt{ = cons } (\top_{bool} * \top_{blist}) \\ \quad \texttt{and } bem \texttt{ = nil } (\textit{runit})\end{array}$$

and value constructors are lazy, then bev & bod contains the infinite value

$$\texttt{cons (true (), cons (true (), ...))},$$

but if value constructors are eager, there are no infinite values and this type is empty. By contrast, the type bem & $bnem$ is empty regardless. Thus, if our type system takes no account of the fact that value constructors in SML are eager, we will have

$$bem \ \& \ bnem \leq bod \ \& \ bev$$

but not

$$bod \ \& \ bev \leq bem \ \& \ bnem.$$

This distinction is an unintuitive nuisance to a programmer who expects value constructors to be eager.

These unnecessary distinctions seem to arise most often for empty recursive types. In this section we define an algorithm that determines when a recursive type is empty if value constructors are call by value. The definition of subtyping for recursive types that appears in the next section ensures that empty recursive types are always subtypes of other recursive types that refine the same ML type. Thus, we will be able to derive

$$bod \ \& \ bev \leq bem \ \& \ bnem.$$

$$\text{RCON-EMPTY:}\quad \frac{rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \overset{\text{def}}{\text{empty}}}{\vdash rc_1 \,\&\, \ldots \,\&\, rc_n \text{ empty}}$$

$$\text{REF-TUPLE-EMPTY:}\quad \frac{\text{for some } i \text{ in } 1 \ldots n \text{ we have} \vdash r_{1i} \wedge \ldots \wedge r_{mi} \text{ empty}}{\vdash (r_{11} * \ldots * r_{1n}) \wedge \ldots \wedge (r_{m1} * \ldots * r_{mn}) \text{ empty}}$$

Figure 3.3: When a Refinement Type is Empty

We will describe the algorithm for determining whether a recursive type is empty in several steps. First in Subsection 3.3.1 we shall postulate a property of refinement type constructors that says whether they are empty. This can easily be extended to a judgement $\vdash r$ empty that says when a refinement type r is empty. We assume that these judgements are consistent in certain ways. Then in Subsection 3.3.2 we shall give declarative inference rules for the judgement that a recursive type is empty, written $D \vdash nr$ empty, where D is an abstract declaration and nr is the recursive type in question. Infinite proofs with these inference rules will be allowed, as they were for the $D \vdash v \in r$ judgement. We also present type inference rules for a relation $D; S \vdash nr$ alg-empty which includes a set S of intersections of recursive type constructors that are presumed empty. Proper use of these inference rules only allows finite proofs, and can be easily read as an algorithm. Then in Subsection 3.3.3 we will prove several properties of these judgements; the most interesting ones are that the algorithmic and declarative are equivalent and that types judged empty actually contain no values.

3.3.1 Emptyness for Refinement Types

We start by assuming that some refinement type constructors are empty. We write the assertion that rc is empty as $rc \overset{\text{def}}{\text{empty}}$. If we assume that certain refinement type constructors are empty, it is straightforward to conclude that certain refinement types are empty. We call the judgement for this $\vdash r$ empty and define it by the rules in Figure 3.3.

For these rules to work properly, we need some consistency between the $rc \overset{\text{def}}{\text{empty}}$ and the $\vdash r$ empty judgements; we can also regard these as consistency conditions on the implicit global environment, as were the assumptions listed in Chapter 2. First, if a value constructor returns something with an empty type, it was given something with an empty type:

Assumption 3.12 (Emptyness Constructor) *If* $rc \overset{\text{def}}{\text{empty}}$ *and* $c \overset{\text{def}}{:} r \hookrightarrow rc$ *then* $\vdash r$ empty.

Also, if a refinement type constructor is empty, any smaller refinement type constructor must also be empty:

Assumption 3.13 (Emptyness Subtyping) *If rc* $\overset{\text{def}}{\text{empty}}$ *and kc* $\overset{\text{def}}{\leq}$ *rc then kc* $\overset{\text{def}}{\text{empty}}$.

These assumptions are sufficient to show that emptyness for refinement types is sound, in the sense that empty refinement types contain no values:

Fact 3.14 (Soundness of Refinment Type Empty) *If* $\vdash r$ empty *and* $\cdot \vdash v :: t$ *and* $r \sqsubset t$ *then we do not have* $\cdot \vdash v : r$.

The proof of this is a straightforward induction on v that we shall omit.

3.3.2 Emptyness for Recursive Types

Emptyness for recursive types is more interesting because we must either allow infinite proofs to get correct behavior in the presence of recursion, or use trails to ensure termination.

For example, using the `rectype` declarations on page 183, we should be able to infer that bev & bod is empty. Suppose, by way of contradiction, that a value has this type; then the value will be in both bev and bod. By the declarations of these types, the outermost constructor of this value must be `cons`, and the argument to `cons` will be in both of the types $\top_{bool} * bod$ and $\top_{bool} * bev$. This can only be the case if there is some value in the type bod & bev. We assume that & for recursive types is commutative, so this is equivalent to the problem we started with. We can either continue to produce an infinite argument, or we can keep track of the set of subproblems already encountered (this set is called a *trail*) so we can observe that we have encountered this problem before and stop. Since there are actually no values with this type, either approach should lead to the conclusion that the type is empty.

If we take the approach of permitting infinite proofs, we get the inference system in Figure 3.4. For this example, the infinite proof tree for $D \vdash bev$ & bod empty is

$$\dfrac{\dfrac{\dfrac{\cdots}{D \vdash bod \;\&\; bev \text{ empty}}\,[\text{NEW-INFER-EMPTY}]}{D \vdash \top_{bool} * bod \;\&\; \top_{bool} * bev \text{ empty}}\,[\text{REC-TUPLE-EMPTY}]}{D \vdash bev \;\&\; bod \text{ empty}}\,[\text{NEW-INFER-EMPTY}]$$

Finding an intuitively meaningful reading is straightforward, with the possible exception of NEW-INFER-EMPTY. Translating it into words yields "If the only way to construct an element of a recursive type nr is by starting with elements of other types that are all empty, then nr is empty.", which seems plausible.

If, instead, we take the approach of using a trail to keep track of the pending subproblems, we get the inference system in Figure 3.5. In this system, the trail is the set S, which

$$\text{NEW-INFER-EMPTY:}\quad \frac{\text{for all } c \text{ and all } rt \text{ such that } \&nrcs \succeq c(nr) \in \overline{D} \text{ we have } D \vdash nr \text{ empty}}{D \vdash \&nrcs \text{ empty}}$$

$$\text{OLD-EMPTY:}\quad \frac{\overset{\text{def}}{\bigwedge} rcs \overset{\text{def}}{\text{empty}}}{D \vdash \&rcs \text{ empty}}$$

$$\text{REC-TUPLE-EMPTY:}\quad \frac{\text{for some } i \text{ in } 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \text{ empty}}{D \vdash (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn}) \text{ empty}}$$

Figure 3.4: Declarative Emptyness for Recursive Types (Greatest Fixed Point)

$$\text{ALG-NEW-ENV-EMPTY:}\quad \frac{\& \; nrcs \in S}{D; S \vdash \&nrcs \text{ alg-empty}}$$

$$\text{ALG-NEW-INFER-EMPTY:}\quad \frac{\text{for all } c \text{ and all } rt \text{ such that } \&nrcs \succeq c(nr) \in \overline{D} \text{ we have } D; S \cup \{\&nrcs\} \vdash nr \text{ alg-empty}}{D; S \vdash \&nrcs \text{ alg-empty}}$$

$$\text{ALG-OLD-EMPTY:}\quad \frac{\overset{\text{def}}{\bigwedge} rcs \overset{\text{def}}{\text{empty}}}{D; S \vdash \&rcs \text{ alg-empty}}$$

$$\text{ALG-REC-TUPLE-EMPTY:}\quad \frac{\text{for some } i \text{ in } 1 \ldots n \text{ we have } D; S \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \text{ alg-empty}}{D; S \vdash (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn}) \text{ alg-empty}}$$

Figure 3.5: Algorithmic Emptyness for Recursive Types

contains the intersections of recursive type constructors that we are already attempting to prove empty. For example, the following derivation is a proof that $bev \;\&\; bod$ is empty:

$$\dfrac{\dfrac{\dfrac{}{D; \{bev \;\&\; bod\} \vdash bod \;\&\; bev \text{ alg-empty}}\,[\text{ALG-NEW-ENV-EMPTY}]}{D; \{bev \;\&\; bod\} \vdash \top_{bool} * bod \;\&\; \top_{bool} * bev \text{ alg-empty}}\,[\text{ALG-REC-TUPLE-EMPTY}]}{D; \{\} \vdash bev \;\&\; bod \text{ alg-empty}}\,[\text{ALG-NEW-INFER-EMPTY}]$$

This inference system can easily be read as an algorithm. Simply try to construct a derivation starting at the root with an empty trail, and use ALG-NEW-ENV-EMPTY instead of

ALG-NEW-INFER-EMPTY whenever a choice arises. Informally speaking, the algorithm for inferring $D; S \vdash nr$ alg-empty is sure to terminate because at each step either S stays the same and nr gets smaller, or S gets larger. Since D is finite and S contains intersections of sets of recursive type constructors mentioned in D, the largest possible S is finite. Formalizing this requires introducing two new definitions: a measure of the size of nr that we shall call depth(nr) and the maximal value of S which we call emptyU(D).

Because of the ALG-REC-TUPLE-EMPTY rule, we cannot say that if S remains constant, nr is replaced by a subterm of itself. For instance, given the problem $D; S \vdash (tt * ff)$ & $(ff * ff)$ alg-empty, we would examine the subproblems $D; S \vdash tt$ & ff alg-empty and $D; S \vdash ff$ & ff alg-empty. Neither of these recursive types appear literally within $(tt * ff)$ & $(ff * ff)$. They are smaller in the sense that their printed representation is smaller, but this is awkward to reason about. Instead we regard the recursive type as a tree, and think in terms of the height of the tree. Since & for recursive types is assumed to be idempotent, we have to give the same "height" to both tt & tt and tt; thus we call it "depth" to distinguish it from the ordinary notion of tree height, and we define it so intersection operators do not increase the measure of a recursive type:

Definition 3.15 (Depth of a Recursive Type) *We define the* depth *of a recursive type by the equations*

$$\begin{aligned}
\text{depth}(\&nrs) &= \max\{\text{depth}(nr) \mid nr \in nrs\} \\
\text{depth}(r \to nr) &= \text{depth}(nr) + 1 \\
\text{depth}(rt_1 * \ldots * rt_n) &= \max\{\text{depth}(nr_i \mid i \in 1 \ldots n)\} + 1 \\
\text{depth}(rc) &= 0 \\
\text{depth}(nrc) &= 0.
\end{aligned}$$

By Condition 3.2 (New Recursive Type Constructors Defined) on page 178, all recursive type constructors that can appear in S are defined in D. Thus we can define the universe from which S is chosen as all possible subsets of the types defined in D:

Definition 3.16 *Define* emptyU(D) *to be* $\{\&nrcs \mid$ *all* $nrc \in nrcs$ *are defined in* $D\}$.

With these definitions, we can say that the natural algorithm derived from the rules in Figure 3.5 terminates because the pair $(\text{emptyU}(D) - S, \text{depth}(nr))$ always lexicographically decreases. Not surprisingly, this measure will also ensure that some induction proofs below make progress.

3.3.3 Properties of Empty

We shall show that the algorithmic and declarative versions of emptyness inference are equivalent, and that types judged empty actually contain no values.

Fact 3.17 (Algorithmic Emptyness Strengthening) *If* $D; S_1 \vdash$ *nr* alg-empty *then*

$$D; S_1 \cup S_2 \vdash \textit{nr} \text{ alg-empty.}$$

Proving this is a trivial induction on the derivation of the hypothesis. The derivation of $D; S_1 \cup S_2 \vdash nr$ alg-empty has the same shape as the derivation of $D; S_1 \vdash nr$ alg-empty; the only difference is that in the former derivation all of the trails are larger.

Lemma 3.18 (Empty Eliminable Assumptions) *If*

$$D; \{\} \vdash \&\textit{nkcs} \text{ alg-empty}$$

and

$$D; S \cup \{\&\textit{nkcs}\} \vdash \textit{nr} \text{ alg-empty}$$

then

$$D; S \vdash \textit{nr} \text{ alg-empty.}$$

Proof: By induction on the derivation of $D; S \cup \{\&nkcs\} \vdash nr$ alg-empty.

Case: ALG-NEW-ENV-EMPTY, $nr = \&nkcs$ Applying Fact 3.17 (Algorithmic Emptyness Strengthening) on page 188 to $D; \{\} \vdash \&nkcs$ alg-empty gives $D; S \vdash \&nkcs$ alg-empty, which is our conclusion.

Case: ALG-NEW-ENV-EMPTY, $nr \neq \&nkcs$ Then $nr \propto \&nrcs$ where the premise of ALG-NEW-ENV-EMPTY is $\&nrcs \in S \cup \{\&nkcs\}$. Since $nr \neq \&nkcs$, this implies $\&nrcs \in S$, and ALG-NEW-ENV-EMPTY gives $D; S \vdash \&nrcs$ alg-empty, which is our conclusion.

Case: ALG-NEW-INFER-EMPTY Then $nr \propto \&nrcs$ and the premise of ALG-NEW-INFER-EMPTY must be

$$\begin{array}{c}\text{for all } c \text{ and all } nk \text{ such that } \&nrcs \succeq c(nk) \in \overline{D} \text{ we have} \\ D; S \cup \{\&nkcs, \&nrcs\} \vdash nk \text{ alg-empty.}\end{array}$$

By induction hypothesis,

$$\begin{array}{c}\text{for all } c \text{ and all } nk \text{ such that } \wedge nrcs \succeq c(nk) \in \overline{D} \text{ we have} \\ D; S \cup \{\&nrcs\} \vdash nk \text{ alg-empty.}\end{array}$$

and ALG-NEW-INFER-EMPTY gives our conclusion.

Case: ALG-OLD-EMPTY Then $nr \propto \&rcs$ and the premise of ALG-OLD-EMPTY is $\vdash \overset{\text{def}}{\wedge} rcs$ empty. ALG-OLD-EMPTY gives our conclusion.

Case: ALG-REC-TUPLE-EMPTY Then $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ and the premise of ALG-REC-TUPLE-EMPTY is

$$\text{for some } i \text{ in } 1 \ldots n \text{ we have } D; S \cup \{\&nkcs\} \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \text{ alg-empty.}$$

By induction,

$$\text{for some } i \text{ in } 1 \ldots n \text{ we have } D; S \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \text{ alg-empty,}$$

and ALG-REC-TUPLE-EMPTY gives our conclusion. □

Now we can show that the algorithmic and declarative versions of emptyness inference are equivalent. We have separate proofs showing each is at least as strong as the other. The first proof is by induction on the pair $(\text{emptyU}(D) - S, \text{depth}(nr))$ ordered lexicographically; the second is the first co-induction in this chapter.

Theorem 3.19 (Emptyness Consistency I) *If* $D \vdash$ *nr* empty *and for all* &*nrcs* $\in S$ *we have* $D \vdash$ &*nrcs* empty *then* $D; S \vdash$ *nr* alg-empty.

Proof: By induction on the pair $(\text{emptyU}(D) - S, \text{depth}(nr))$, ordered lexicographically. The declarative emptyness rules constrain the form of nr, so we have the following cases:

Case: $nr \propto \&nrcs$ If $nr \in S$, then ALG-NEW-ENV-EMPTY gives our conclusion.

Otherwise, the last inference of $D \vdash nr$ empty must be NEW-INFER-EMPTY with the premise

$$\text{for all } c \text{ and all } nk \text{ such that } \&nrcs \succeq c(nk) \in \overline{D} \text{ we have } D \vdash nk \text{ empty.}$$

Combining the two hypotheses of this theorem,

$$\text{for all } \&nkcs \in S \cup \{\&nrcs\} \text{ we have } D \vdash \&nkcs \text{ empty.}$$

The induction hypothesis gives

$$\begin{array}{l}\text{for all } c \text{ and all } nk \text{ such that } \&nrcs \succeq c(nk) \in \overline{D} \text{ we have} \\ \quad D; S \cup \{\&nrcs\} \vdash nk \text{ alg-empty,}\end{array}$$

and ALG-NEW-INFER-EMPTY gives our conclusion.

Case: $nr \propto \&rcs$ Then the last inference of $D \vdash nr$ empty is OLD-EMPTY with the premise $\overset{\text{def}}{\wedge} rcs$ $\overset{\text{def}}{\text{empty}}$ and ALG-OLD-EMPTY gives $D; S \vdash nr$ alg-empty, which is our conclusion.

Case: $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ Then the last inference of $D \vdash nr$ empty is REC-TUPLE-EMPTY with the premise

for some i in $1 \ldots n$ we have $D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}$ empty.

The induction hypothesis gives

for some i in $1 \ldots n$ we have $D; S \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}$ alg-empty.

and ALG-REC-TUPLE-EMPTY gives our conclusion. □

Theorem 3.20 (Emptyness Consistency II) *If* $D; \{\} \vdash$ *nr* alg-empty *then* $D \vdash$ *nr* empty.

Proof: By co-induction. Take D to be fixed, and let F be the natural encoding of the rules in Figure 3.4 as a function from sets of recursive types to sets of recursive types. Let $Q = \{nr \mid D; \{\} \vdash nr \text{ alg-empty}\}$; thus our goal is to show $Q \subset \text{gfp}(F)$. By co-induction, it suffices to show $Q \subset F(Q)$. Let nr be an element of Q. We will show by cases on nr that nr is in $F(Q)$ as well.

Case: $nr \propto \&nrcs$ The last inference of $D; \{\} \vdash nr$ alg-empty must be ALG-NEW-INFER-EMPTY with the premise

for all c and all nk such that $\&nrcs \succeq c(nk) \in \overline{D}$ we have $D; \{\&nrcs\} \vdash nk$ alg-empty.

By Lemma 3.18 (Empty Eliminable Assumptions) on page 188 we have

for all c and all nk such that $\&nrcs \succeq c(nk) \in \overline{D}$ we have $D; \{\} \vdash nk$ alg-empty

and the definition of Q gives

for all c and all nk such that $\&nrcs \succeq c(nk) \in \overline{D}$ we have $nk \in Q$.

By NEW-INFER-EMPTY, this implies $\&nrcs \in F(Q)$, which is what we wanted to show.

The next two cases are trivial, but they are also short, so we include them for completeness.

Case: $nr \propto \&rcs$ The last inference of $D; \{\} \vdash nr$ alg-empty must be ALG-OLD-EMPTY with the premise $\overset{\text{def}}{\wedge} rcs \overset{\text{def}}{\text{empty}}$. By OLD-EMPTY, this implies $nr \in F(Q)$, which is our conclusion.

Case: $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ Then the last inference of $D; \{\} \vdash nr$ alg-empty is ALG-REC-TUPLE-EMPTY with the premise

for some i in $1 \ldots n$ we have $D; \{\} \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}$ alg-empty.

The definition of Q gives

$$\text{for some } i \text{ in } 1 \ldots n \text{ we have } nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \in Q.$$

REC-TUPLE-EMPTY then gives $nr \in F(Q)$, which is our conclusion. □

We shall say that the last inferences of a derivation have some property if every path from the root of the derivation starts with one or more inferences that have that property. For an example, see the first case of the following proof.

Theorem 3.21 (Soundness of Empty) *We never have* $D \vdash nr$ empty *and* $D \vdash v \in nr$.

Proof: By induction on v.

Case: $v \propto c\ \ v'$ where c is new Then the last inferences of the derivation of $D \vdash v \in nr$ must be AND-RECVALUE and NEW-RC-RECVALUE, so $nr \propto \&nrcs$. The last inference of $D \vdash nr$ empty must be NEW-INFER-EMPTY with the premises

$$\text{for all } c \text{ and all } nk \text{ such that } \&nrcs \succeq c(nk) \in \overline{D} \text{ we have } D \vdash nk \text{ empty.} \tag{3.1}$$

The premises of AND-RECVALUE and NEW-RC-RECVALUE leading up to $D \vdash v \in nr$ must be

$$\begin{array}{l}\text{for all } nrc \in nrcs \text{ there is a } np_{nrc} \text{ such that} \\ \quad nrc \succeq c(np_{nrc}) \in \overline{D} \text{ and} \\ \quad D \vdash v' \in np_{nrc}.\end{array} \tag{3.2}$$

By definition of intersection membership,

$$\&nrcs \succeq c(\&\{np_{nrc} \mid nrc \in nrcs\}) \in \overline{D}$$

thus (3.1) gives

$$D \vdash \&\{np_{nrc} \mid nrc \in nrcs\} \text{ empty.}$$

Applying AND-RECVALUE to (3.2) gives

$$D \vdash v' \in \&\{np_{nrc} \mid nrc \in nrcs\}.$$

The induction hypothesis applied to the last two displayed formulae yields our contradiction.

Case: $v \propto c\ \ v'$ where c is old Then the last inferences of $D \vdash v \in nr$ must be AND-RECVALUE and OLD-RC-RECVALUE, so $nr \propto \&rcs$ and for all rc in rcs we have $\cdot \vdash c\ \ v' : rc$. By AND-INTRO-TYPE we have

$$\cdot \vdash c\ \ v' : \&rcs.$$

The last inference of $D \vdash nr$ empty must be OLD-EMPTY with the premise $\overset{\text{def}}{\wedge}\ rcs\ \overset{\text{def}}{\text{empty}}$; RCON-EMPTY then gives $\vdash \&rcs$ empty and by Fact 3.14 (Soundness of Refinement Type Empty) on page 185 we do not have

$$\cdot \vdash c\ \ v' : \&rcs.$$

This is our contradiction.

Case: $v \propto (v_1, \ldots, v_n)$ Then the last inferences of $D \vdash v \in nr$ must be AND-RECVALUE and TUPLE-RECVALUE, so $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$. Thus the last inference of $D \vdash nr$ empty must be REC-TUPLE-EMPTY with the premise

$$\text{for some } i \text{ in } 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \text{ empty.} \tag{3.3}$$

The premises of AND-RECVALUE and TUPLE-RECVALUE leading up to $D \vdash v \in nr$ must be

$$\text{for all } i \text{ in } 1 \ldots n \text{ and all } j \text{ in } 1 \ldots m \text{ we have } D \vdash v_i \in nr_{ji}$$

and AND-RECVALUE gives

$$\text{for all } i \text{ in } 1 \ldots n \text{ we have } D \vdash v_i \in nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}. \tag{3.4}$$

Our induction hypothesis applied to (3.3) and (3.4) gives our contradiction.

Case: $v \propto \texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ e$ Then the last inferences of $D \vdash v \in nr$ must be ABS-RECVALUE and AND-RECVALUE, so $nr \propto r_1 \to nr_1 \,\&\, \ldots \,\&\, r_n \to nr_n$ and there is no way to infer $D \vdash nr$ empty. □

The intersection of an empty recursive type and any other recursive type is also empty, if it the intersection is well-formed. We include the proof to give another example of an ordinary co-induction proof, slightly more complex than Theorem 3.20 (Emptyness Consistency II) on page 190.

Theorem 3.22 (Empty Intersection) *If $D \vdash nr$* empty *and $D \vdash nr \,\&\, nk \sqsubset t$ then $D \vdash nr \,\&\, nk$* empty.

Proof: By co-induction. Take D as fixed, and let F be the natural description of the rules in Figure 3.4 as a function from sets of recursive types to sets of recursive types. Let $Q = \{nr \,\&\, nk \mid D \vdash nr$ empty and there is a t such that $D \vdash nr \,\&\, nk \sqsubset t\}$. We need to show $Q \subset \text{gfp}(F)$; by co-induction, it suffices to show $Q \subset F(Q)$. Let $nr \,\&\, nk$ be an arbitrary element of Q; it suffices to show that $nr \,\&\, nk \in F(Q)$. We take cases on the form of nr.

Case: $nr \propto \&nrcs$ Then by definition of Q we have $D \vdash \&nrcs$ empty and $D \vdash (\&nrcs) \,\&\, nk \sqsubset t$. We can only infer the latter if $nk \propto \&nkcs$. The last inference of $D \vdash \&nrcs$ empty must be NEW-INFER-EMPTY with the premise

$$\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \text{ we have } D \vdash nr' \text{ empty.}$$

Let c and np be given such that $\&(nrcs \cup nkcs) \succeq c(np) \in \overline{D}$. By definition of intersection membership, we can write np as $nr' \,\&\, nk'$ where $\&nrcs \succeq c(nr')$. By Fact 3.8 (Intersection Refines) on page 179, there is a u such that $D \vdash nr' \,\&\, nk' \sqsubset u$, so the definition of Q gives

$$\text{for all } c \text{ and all } np \text{ such that } \&(nrcs \cup nkcs) \succeq c(np) \in \overline{D} \text{ we have } np \in Q$$

Thus, by NEW-INFER-EMPTY, $\&(nrcs \cup nkcs) \in F(Q)$, which is our conclusion.

Case: $nr \propto \&rcs$ Since $D \vdash nr \;\&\; nk \sqsubset t$, we know $nk \propto \&kcs$. The last inference of $D \vdash nr$ empty must be OLD-EMPTY with the premise $\overset{\text{def}}{\wedge}\, rcs$ $\overset{\text{def}}{\text{empty}}$. Simple reasoning about $\overset{\text{def}}{\wedge}$ gives $(\overset{\text{def}}{\wedge}\, rcs) \overset{\text{def}}{\wedge} (\overset{\text{def}}{\wedge}\, kcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge}\, rcs)$, so Assumption 3.13 (Emptyness Subtyping) on page 185 gives $(\overset{\text{def}}{\wedge}\, rcs) \overset{\text{def}}{\wedge} (\overset{\text{def}}{\wedge}\, kcs)$ $\overset{\text{def}}{\text{empty}}$. OLD-EMPTY then gives $(\&rcs)\&(\&kcs) \in F(Q)$, which is our conclusion.

Case: $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ Since $D \vdash nr \;\&\; nk \sqsubset t$, we must have $nk \propto (nk_{11} * \ldots * nk_{1n}) \,\&\, \ldots \,\&\, (nk_{q1} * \ldots * nk_{qn})$. The last inference of $D \vdash nr$ empty must be REC-TUPLE-EMPTY with, for some i, the premise $D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}$ empty. Simple reasoning about recursive refinement types gives a u such that $D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \,\&\, nk_{1i} \,\&\, \ldots nk_{qi} \sqsubset u$. The definition of Q then gives $nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \,\&\, nk_{1i} \,\&\, \ldots nk_{qi} \in Q$, then REC-TUPLE-EMPTY gives $nr \;\&\; nk \in F(Q)$, which is our conclusion. □

3.4 Subtyping

The type inference system for subtyping recursive types is similar to the system in the previous section for inferring when a type is empty. For subtyping we have two similar systems, one declarative using a greatest fixed point and no trail, and one algorithmic using a least fixed point and a trail. In this case a trail is a set with elements of the form $(\&nrcs, \&nkcs)$; each element represents the assertion that we are already working on the problem $D \vdash \&nrcs \leq \&nkcs$.

The declarative system is described in Figure 3.6, and the algorithmic system is Figure 3.7. The OLD-RECSUB and TUPLE-RECSUB rules are self-evident; explanations of the other two follow.

One way to understand the NEW-INFER-RECSUB rule is by walking through a sketch of that case of Theorem 3.34 (Recursive Subtype Soundness) on page 204. Suppose some value $c\ v$ is in $\&nrcs$. Then there must be some definition of $\&nrcs$ of the form $c(nr)$ where v is in nr. If nr is empty, then we have a contradiction and we are done. Otherwise, if there is a definition of $\&nkcs$ of the form $c(nk)$ for some nk larger than nr, then v is in nk and $c\ v$ is in $\&nkcs$.

Although this rule is sound, it could be stronger. For example, consider the declaration

```
datatype d = C of bool
rectype d1 = C (⊤bool)
    and d2 = C (tt) | C (ff)
```

$$\text{NEW-INFER-RECSUB:}\quad \frac{\begin{array}{c} D \vdash \ \&\, nrcs \sqsubset tc \\ D \vdash \ \&\, nkcs \sqsubset tc \\ \text{for all } c \text{ and all } nr \text{ such that } \&nrcs \succeq c(nr) \in \overline{D} \\ \text{either } D \vdash nr \text{ empty} \\ \text{or there is a } nk \text{ such that } \&nkcs \succeq c(nk) \in \overline{D} \text{ and } D \vdash nr \leq nk \end{array}}{D \vdash \&nrcs \leq \&nkcs}$$

$$\text{ARROW-RECSUB:}\quad \frac{\begin{array}{c} D \vdash \ \&\, \{r_i \to nr_i \mid i \in 1 \ldots n\} \sqsubset t \\ \text{for } j \in 1 \ldots m \text{ we have } D \vdash \&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \leq nk_j \end{array}}{D \vdash \&\{r_i \to nr_i \mid i \in 1 \ldots n\} \leq \&\{k_j \to nk_j \mid j \in 1 \ldots m\}}$$

$$\text{OLD-RECSUB:}\quad \frac{(\overset{\text{def}}{\wedge}\, rcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge}\, kcs)}{D \vdash \&rcs \leq \&kcs}$$

$$\text{TUPLE-RECSUB:}\quad \frac{\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \leq nk_{1i} \,\&\, \ldots \,\&\, nk_{qi}}{\begin{array}{c} D \vdash (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn}) \leq \\ (nk_{11} * \ldots * nk_{1n}) \,\&\, \ldots \,\&\, (nk_{q1} * \ldots * nk_{qn}) \end{array}}$$

Figure 3.6: Declarative Rules for Recursive Subtyping (Greatest Fixed Point)

With this declaration, all values in d_1 are also in d_2, but if we convert this declaration into an abstract declaration D we cannot infer $D \vdash d_1 \leq d_2$. The cause of this is that $\top_{bool}$ is less than the union of tt and ff, but it is not less than either tt or ff taken individually. Since it is possible to decide whether the language recognized by one regular tree automaton is a subset of the language recognized by another ([GS84]), and `rectype` statements that do not contain "$\to$" are essentially descriptions of regular tree automata, it is in principle possible to make a practical system that is complete in the first-order case.

The ARROW-RECSUB rule is motivated by Lemma 2.83 (i Gives an Upper Bound) on page 111. It would probably be possible to take the approach of Chapter 2 and have simple axioms defining recursive type inference and then restructure the system completely to find a practical algorithm that uses ARROW-RECSUB, but such an analysis might be as long as Chapter 2. Our grammar does not admit & with zero arguments, so this rule does not apply if any of the sets mentioned are empty. For example, suppose D includes the usual definitions of refinements of $bool$ and we are trying to prove the false assertion $D \vdash tt \to ff \leq ff \to tt$; then one of the premises would have to be $D \vdash \wedge\{\} \leq tt$, which is malformed.

The algorithmic and the declarative systems are consistent in the same sense the systems for emptyness were consistent. The proof is entirely analogous to the proof that the systems for emptyness are consistent. We start by establishing that we can manipulate the trail:

Fact 3.23 (Subtype Strengthening) *If $S' \subset S$ and $D; S' \vdash nr \leq nk$ then $D; S \vdash nr \leq nk$.*

ALG-NEW-ENV-RECSUB:
$$\frac{D \vdash \ \&\ nrcs \sqsubset tc \quad D \vdash \ \&\ nkcs \sqsubset tc \quad (\&nrcs, \&nkcs) \in S}{D; S \vdash \&nrcs \leq \&nkcs}$$

ALG-NEW-INFER-RECSUB:
$$\frac{\begin{array}{c} D \vdash \ \&\ nrcs \sqsubset tc \quad D \vdash \ \&\ nkcs \sqsubset tc \\ \text{for all } c \text{ and all } nr \text{ such that } \&nrcs \succeq c(nr) \in \overline{D} \\ \text{either } D \vdash nr \text{ empty} \\ \text{or there is a } nk \text{ such that} \\ \&nkcs \succeq c(nk) \in \overline{D} \text{ and } D; S \vdash nr \leq nk \end{array}}{D; S \vdash \&nrcs \leq \&nkcs}$$

ALG-ARROW-RECSUB:
$$\frac{\begin{array}{c} D \vdash \ \&\ \{r_i \rightarrow nr_i \mid i \in 1 \ldots n\} \sqsubset t \\ \text{for } j \in 1 \ldots m \text{ we have} \\ D; S \vdash \&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \leq nk_j \end{array}}{D; S \vdash \&\{r_i \rightarrow nr_i \mid i \in 1 \ldots n\} \leq \&\{k_j \rightarrow nk_j \mid j \in 1 \ldots m\}}$$

ALG-OLD-RECSUB:
$$\frac{(\overset{\text{def}}{\wedge}\ rcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge}\ kcs)}{D; S \vdash \&rcs \leq \&kcs}$$

ALG-TUPLE-RECSUB:
$$\frac{\begin{array}{c} \text{for } i \in 1 \ldots n \text{ we have} \\ D; S \vdash nr_{1i}\ \&\ \ldots\ \&\ nr_{mi} \leq nk_{1i}\ \&\ \ldots\ \&\ nk_{qi} \end{array}}{\begin{array}{c} D; S \vdash (nr_{11} * \ldots * nr_{1n})\ \&\ \ldots\ \&\ (nr_{m1} * \ldots * nr_{mn}) \leq \\ (nk_{11} * \ldots * nk_{1n})\ \&\ \ldots\ \&\ (nk_{q1} * \ldots * nk_{qn}) \end{array}}$$

Figure 3.7: Algorithmic Rules for Recursive Subtyping

The proof is a simple induction on the derivation of $D; S' \vdash nr \leq nk$.

Fact 3.24 (Subtype Eliminable Assumptions) *If* $D; \{\} \vdash \&nrcs \leq \&nkcs$ *and* $D; S \cup \{(\&nrcs, \&nkcs)\} \vdash nr \leq nk$ *then* $D; S \vdash nr \leq nk$.

The proof is by induction on the derivation of $D; S \cup \{(\&nrcs, \&nkcs)\} \vdash nr \leq nk$.

As we did for the rules for emptyness, we must define a universe of all possible members of the trail:

Definition 3.25 *Define* subtypeU(D) *to be*

$$\{(nr, nk) \mid nr \in \text{emptyU}(D) \text{ and } nk \in \text{emptyU}(D)\}.$$

and continue with separate proofs for the if and only if cases:

Fact 3.26 (Recursive Subtype Consistency I)
If $D \vdash$ *nr* $\leq$ *nk and for all* $(\&nrcs, \&nkcs) \in S$ *we have* $D \vdash \&nrcs \leq \&nkcs$ *then* $D; S \vdash$ *nr* $\leq$ *nk.*

Proof is by induction on the pair $(\text{subtypeU}(D) - S, \text{depth}(nr))$, ordered lexicographically.

Fact 3.27 (Recursive Subtype Consistency II) *If* $D; \{\} \vdash$ *nr* $\leq$ *nk then* $D \vdash$ *nr* $\leq$ *nk.*

Proof is by co-induction, and is similar to the proof of Theorem 3.20 (Emptyness Consistency II) on page 190.

An analogue for Theorem 2.21 (Subtypes Refine) on page 36 holds for recursive types:

Fact 3.28 (Recursive Subtypes Refine) *If* $D \vdash$ *nr* $\leq$ *nk then there is a* t *such that* $D \vdash$ *nr* $\sqsubset t$ *and* $D \vdash$ *nk* $\sqsubset t$.

Proof is by induction on the depth of nr.

For refinement types, we explicitly assumed that intersection is a greatest lower bound. For recursive types, we must prove it. The proof is not very interesting; it is included because it is a proof about recursive subtyping that has no analog in Section 3.3.

Lemma 3.29 (Recursive Intersection Lower Bound) *If* $D \vdash$ *nr* $\leq$ *nk and* $D \vdash$ *nr*$\&$*np* $\sqsubset$ t *then* $D \vdash$ *nr* $\&$ *np* $\leq$ *nk.*

Proof: By co-induction. Take D as fixed, and let F be an encoding of the declarative subtype inference system in Figure 3.6 as a function from sets of pairs of recursive types to sets of pairs of recursive types. Let

$$Q = \{(nr \,\&\, np, nk) \mid D \vdash nr \leq nk \text{ and for some } t \text{ we have } D \vdash nr \,\&\, np \sqsubset t\}.$$

Then our goal is to show $Q \subset \text{gfp}(F)$, and by co-induction, it suffices to show $Q \subset F(Q)$. Let $(nr \,\&\, np, nk)$ be any element of Q; if we can show $(nr \,\&\, np, nk) \in F(Q)$, we are done. We must have $D \vdash nr \,\&\, np \sqsubset t$, so nr must have one of the following forms:

Case: $nr \propto \&nrcs$ Then the last inference of $D \vdash nr \leq nk$ must be NEW-INFER-RECSUB, so $nk \propto \&nkcs$ and the premises of NEW-INFER-RECSUB are

$$D \vdash \&\, nrcs \sqsubset tc$$
$$D \vdash \&\, nkcs \sqsubset tc$$

where tc is new and

$$\begin{array}{l}\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \\ \text{either } D \vdash nr' \text{ empty} \\ \text{or there is a } nk' \text{ such that } \&nkcs \succeq c(nk') \in \overline{D} \text{ and } D \vdash nr' \leq nk'\end{array} \tag{3.5}$$

Let c and np'' be given such that $(\&nrcs)$ & $(\&npcs) \succeq c(np'') \in \overline{D}$. By definition of intersection membership, $np'' \propto nr'$ & np' for some nr' such that $\&nrcs \succeq c(nr') \in \overline{D}$. By Fact 3.8 (Intersection Refines) on page 179, there is a u such that

$$D \vdash nr' \,\&\, np' \sqsubset u \tag{3.6}$$

By (3.5) we have the following cases:

SubCase: $D \vdash nr'$ empty By Theorem 3.22 (Empty Intersection) on page 192 we have $D \vdash nr'$ & np' empty.

SubCase: otherwise Then by (3.5) there is a nk' such that $\&nkcs \succeq c(nk') \in \overline{D}$ and $D \vdash nr' \leq nk'$. By definition of Q, this implies $(nr' \,\&\, np', nk') \in Q$.

End SubCase

Summarizing,

$$\begin{array}{l}\text{for all } c \text{ and all } np'' \text{ such that } (\&nrcs) \,\&\, (\&npcs) \succeq c(np'') \in \overline{D} \text{ we have} \\ \text{either } D \vdash np'' \text{ empty} \\ \text{or there is a } nk' \text{ such that } \&nkcs \succeq c(nk') \in \overline{D} \text{ and } (np'', nk') \in Q\end{array}$$

Thus NEW-INFER-RECSUB gives $((\&nrcs) \,\&\, (\&npcs), nk) \in F(Q)$, which is what we wanted to show.

Case: $nr \propto \&\{r_i \to nr_i \mid i \in 1 \ldots n\}$ Then the last inference of $D \vdash nr \leq nk$ is ARROW-RECSUB and $nk = \&\{k_i \to nk_i \mid i \in 1 \ldots m\}$. The premises of ARROW-SUB include

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } D \vdash \&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \leq nk_j.$$

The last inferences of $D \vdash nr \,\&\, np \sqsubset t$ must be AND-RECREFINES and ARROW-RECREFINES so $np \propto \&\{r_i \to nr_i \mid i \in n+1 \ldots q\}$ and $t \propto t_1 \to t_2$ and the premises of ARROW-RECREFINES are

$$\text{for } i \in 1 \ldots q \text{ we have } r_i \sqsubset t_1$$

and

$$\text{for } i \in 1 \ldots q \text{ we have } D \vdash nr_i \sqsubset t_2.$$

Using AND-RECREFINES gives

$$\text{for } j \in 1 \ldots m \text{ we have } D \vdash \,\&\, \{nr_i \mid i \in 1 \ldots q \text{ and } k_j \leq r_i\} \sqsubset t_2.$$

The definition of Q gives

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } (\&\{nr_i \mid i \in 1 \ldots q \text{ and } k_j \leq r_i\}, nk_j) \in Q$$

and then ARROW-RECSUB gives $(nr \;\&\; np, nk) \in F(Q)$, which is our conclusion.

Case: $nr \propto \&rcs$ Then the last inference of $D \vdash nr \leq nk$ must be OLD-RECSUB, where $nk \propto \&kcs$ and the premise of OLD-RECSUB is $(\overset{\text{def}}{\wedge} rcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge} kcs)$. The last inferences of $D \vdash nr \;\&\; np \sqsubset t$ are AND-RECREFINES and OLD-RECREFINES, so $np \propto \&pcs$ and $t \propto tc$ where tc is old and the premises of OLD-RECREFINES include

$$\text{for } rc \in rcs \cup pcs \text{ we have } rc \overset{\text{def}}{\sqsubset} tc.$$

By Assumption 2.16 ($\overset{\text{def}}{\wedge}$ defined) on page 34, $\overset{\text{def}}{\wedge}(rcs \cup pcs)$ is defined, and by Assumption 2.17 ($\overset{\text{def}}{\wedge}$ Elim) on page 34 we have $\overset{\text{def}}{\wedge}(rcs \cup pcs) \overset{\text{def}}{\leq} \overset{\text{def}}{\wedge} kcs$. The OLD-RECSUB gives $(\overset{\text{def}}{\wedge}(rcs \cup pcs), \overset{\text{def}}{\wedge} kcs) \in Q$, which is our conclusion.

Case: $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ Then the last inference of $D \vdash nr \leq nk$ is TUPLE-RECSUB, so $nk \propto (nk_{11} * \ldots * nk_{1n}) \,\&\, \ldots \,\&\, (nk_{q1} * \ldots * nk_{qn})$ and the premises of TUPLE-RECSUB are

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \leq nk_{1i} \,\&\, \ldots \,\&\, nk_{qi}.$$

The last inferences of $D \vdash nr \;\&\; np \sqsubset t$ must be AND-RECREFINES and TUPLE-RECREFINES, so $t \propto t_1 * \ldots * t_n$ and $np \propto (nr_{(m+1)1} * \ldots * nr_{(m+1)n}) \,\&\, \ldots \,\&\, (nr_{z1} * \ldots * nr_{zn})$ and the premises of TUPLE-RECREFINES must be

$$\text{for } i \in 1 \ldots n \text{ and } j \in 1 \ldots z \text{ we have } D \vdash nr_{ji} \sqsubset t_i.$$

AND-RECREFINES then gives

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{zi} \sqsubset t_i,$$

and the definition of Q then gives

$$\text{for } i \in 1 \ldots n \text{ we have } (nr_{1i} \,\&\, \ldots \,\&\, nr_{zi}, nk_{1i} \,\&\, \ldots \,\&\, nk_{qi}) \in Q.$$

TUPLE-RECSUB then gives $(nr \;\&\; np, nk) \in F(Q)$, which is our conclusion. □

Intersection is also a greatest lower bound for recursive types.

Fact 3.30 (Recursive Intersection Greatest) *If $D \vdash nr \leq nk$ and $D \vdash nr \leq np$ then $D \vdash nr \leq nk \;\&\; np$.*

The proof is a straightforward co-induction, and we omit it.

All recursive types that refine some ML type are subtypes of themselves, but to make the co-induction go through we must first instead prove a stronger assertion:

Lemma 3.31 (Self Recsub) *If $D \vdash \&nrs \sqsubset t$ then for any $nks \subset nrs$ we have $D \vdash \&nrs \leq \&nks$.*

Proof: By co-induction. Take D as fixed and let F be the natural encoding of the inference rules in Figure 3.6 as a function from pairs of recursive types to pairs of recursive types. Let $Q = \{(\&nrs, \&nks) \mid D \vdash \& nrs \sqsubset t \text{ and } nks \subset nrs\}$. Then our goal is to show $Q \subset \text{gfp}(F)$, and by co-induction it suffices to show $Q \subset F(Q)$. Proof is by cases on $\&nrs$.

The most natural statement of this theorem would only allow nrs and nks to be identical, each with exactly one element. The case for arrow types required strengthening the co-induction hypothesis to include the possibility that nrs contains more than one element. Once we allow nrs to contain more than one element, the case for recursive type constructors required including the possibility that nks has more than one element.

Case: $\&nrs = \&nrcs$ Then $\&nks \propto \&nkcs$. Let c and nr' such that $\&nrcs \succeq c(nr') \in \overline{D}$ be given. By definition of intersection membership, $nr' \propto \&nrs'$, and there is a $nks' \subset nrs'$ such that $\&nkcs \succeq c(\&nks') \in \overline{D}$. Definition of Q gives $(\&nrs', \&nks') \in Q$. Summarizing this case so far (and adding an otherwise unnecessary disjunction to make the summary have the right form),

> for all c and all $\&nrs'$ such that $\&nrcs \succeq c(\&nrs') \in \overline{D}$
> either $D \vdash \&nrs'$ empty
> or there is a $\&nks'$ such that $\&nkcs \succeq c(\&nks') \in \overline{D}$ and $D \vdash \&nrs' \leq \&nks'$

By NEW-INFER-RECSUB, this implies $(\&nrcs, \&nkcs) \in F(Q)$, which is what we wanted to show.

Case: $\&nrs \propto \&\{r_i \to nr_i \mid i \in 1 \ldots n\}$ Then $\&nks \propto \&\{r_i \to nr_i \mid i \in 1 \ldots m\}$ where $m \leq n$. By SELF-SUB we have

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } r_j \leq r_j$$

and some simple set manipulation and the definition of Q gives

$$\text{for } j \text{ in } 1 \ldots m \text{ we have } (\&\{nr_i \mid i \in 1 \ldots n \text{ and } r_j \leq r_i\}, nr_j) \in Q.$$

ARROW-RECSUB then gives $(\&nrs, \&nks) \in F(Q)$, which is what we wanted to show.

Case: $\&nrs \propto \&rcs$ Then $\&nks \propto \&kcs$, where $kcs \subset rcs$. Simple reasoning about $\overset{\text{def}}{\wedge}$ gives $(\overset{\text{def}}{\wedge} rcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge} kcs)$, and by OLD-RECSUB this implies $(\&nrs, \&nks) \in Q$, which is our conclusion.

Case: $nrs \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{q1} * \ldots * nr_{qn})$ Then $\&nks \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{qn})$ where $q \leq m$. Set manipulation and the definition of Q give

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } (nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}, nr_{1i} \,\&\, \ldots \,\&\, nr_{qi}) \in Q$$

and TUPLE-RECSUB gives $(\&nrs, \&nks) \in F(Q)$, which is our conclusion. □

Now we can move on to prove that recursive subtyping is transitive, which is somewhat more work. Since the subtyping rules mention emptyness, we need to first show that a type smaller than an empty type is also empty. Proving this requires a slightly unusual co-induction; we include the case of the proof that makes the unusualness necessary:

Theorem 3.32 (Empty Transitivity) *If* $D \vdash nk$ empty *and* $D \vdash nr \leq nk$ *then* $D \vdash nr$ empty.

Proof: Take D as fixed, and let F be the natural encoding of the rules for emptyness in Figure 3.4 as a function from sets of recursive types to sets of recursive types. Let $Q' = \{nr \mid \text{there is a } nk \text{ such that } D \vdash nk \text{ empty and } D \vdash nr \leq nk\}$, and let $Q = Q' \cup \text{gfp}(F)$. (This definition of Q allows the first subcase of the first case below to work.) Our theorem is true if $Q' \subset \text{gfp}(F)$, which is true if and only if $Q \subset \text{gfp}(F)$. By co-induction it suffices to show $Q \subset F(Q)$. Proof is by cases on some $nr \in Q$.

If $nr \in \text{gfp}(F)$, then by definition of greatest fixed point, $nr \in F(\text{gfp}(F))$, and by monotonicity of F we have $nr \in F(Q)$. Thus our result always holds if $nr \in \text{gfp}(F)$, and we only need to consider $nr \in Q'$ in the cases below.

Case: $nr \propto \&nrcs$ Then the last inference of $D \vdash nr \leq nk$ must be NEW-INFER-RECSUB, so $nk \propto \&nkcs$ and the premises of NEW-INFER-RECSUB are

$$D \vdash \,\&\, nrcs \sqsubset tc$$
$$D \vdash \,\&\, nkcs \sqsubset tc$$

$$\begin{array}{l}\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \\ \text{either } nr' \in \text{gfp}(F) \\ \text{or there is a } nk' \text{ such that } \&nkcs \succeq c(nk') \in \overline{D} \text{ and } D \vdash nr' \leq nk'.\end{array} \tag{3.7}$$

The last inference of $D \vdash nk$ empty must be NEW-INFER-EMPTY with the premise

$$\text{for all } c \text{ and all } nk' \text{ such that } \&nkcs \succeq c(nk') \in \overline{D} \text{ we have } D \vdash nk' \text{ empty} \tag{3.8}$$

and by NEW-INFER-EMPTY it suffices to show

$$\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \text{ we have } nr' \in Q$$

Let c and nr' such that $\&nrcs \succeq c(nr') \in \overline{D}$ be given. By (3.7), we have these cases:

SubCase: $nr' \in \text{gfp}(F)$ Then, by definition of Q, we have $nr' \in Q$. (This is the step that requires Q to be larger than Q'.)

SubCase: Otherwise Then there is a nk' such that $\&nkcs \succeq c(nk') \in \overline{D}$ and $D \vdash nr' \leq nk'$. By (3.8), this implies $D \vdash nk'$ empty. Definition of Q' gives $nr' \in Q'$, and then definition of Q gives $nr' \in Q$.

End SubCase

Summarizing the above two cases,

$$\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \text{ we have } nr' \in Q$$

NEW-INFER-EMPTY then gives $\&nrcs \in F(Q)$, which is what we wanted to show.

Case: Otherwise The remaining possibilities are all straightforward and are omitted. □

Theorem 3.33 (Subtype Transitivity) *If $D \vdash nr \leq nk$ and $D \vdash nk \leq np$ then $D \vdash nr \leq np$.*

Proof: By co-induction. Take D as fixed, and let F be the encoding of the recursive subtyping relation in Figure 3.6 as a function from pairs of recursive types to pairs of recursive types, and let

$$Q = \{(nr, np) \mid \text{for some } nk \text{ we have } D \vdash nr \leq nk \text{ and } D \vdash nk \leq np\}.$$

We need to prove $Q \subset \text{gfp}(F)$, and by co-induction it suffices to show $Q \subset F(Q)$. Proof is by cases on an nr such that $(nr, np) \in Q$. Delicate use of the fact that intersection for recursive types is a least upper bound is necessary in the case where nr refines an arrow type.

Case: $nr \propto \&nrcs$ Let nk be as given in the definition of Q. Since $D \vdash nr \leq nk$, by NEW-INFER-RECSUB we have $nk \propto \&nkcs$ and the following:

$$D \vdash \ \& \ nrcs \sqsubset tc$$

$$D \vdash \ \& \ nkcs \sqsubset tc$$

$$\begin{array}{l}\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \\ \text{either } D \vdash nr' \text{ empty} \\ \text{or there is a } nk' \text{ such that } \&nkcs \succeq c(nk') \in \overline{D} \text{ and } D \vdash nr' \leq nk'.\end{array} \tag{3.9}$$

Similarly, since $D \vdash nk \leq np$, by NEW-INFER-RECSUB we know $np \propto \&npcs$ and the following:

$$D \vdash \ \& \ npcs \sqsubset tc$$

$$\begin{array}{l}\text{for all } c \text{ and all } nk' \text{ such that } \&nkcs \succeq c(nk') \in \overline{D} \\ \text{either } D \vdash nk' \text{ empty} \\ \text{or there is a } np' \text{ such that } \&npcs \succeq c(np') \text{ and } D \vdash nk' \leq np'\end{array} \tag{3.10}$$

Our goal is to prove $(\&nrcs, \&npcs) \in F(Q)$, and by NEW-INFER-RECSUB it suffices to show

$$\begin{array}{l}\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \\ \text{either } D \vdash nr' \text{ empty} \\ \text{or there is a } np' \text{ such that } \&npcs \succeq c(np') \in \overline{D} \text{ and } (nr', np') \in Q\end{array} \tag{3.11}$$

To prove this, let c and nr' such that $\&nrcs \succeq c(nr') \in \overline{D}$ be given. By (3.9), either $D \vdash nr'$ empty (in which case we are done) or there is a nk' such that $\&nkcs \succeq c(nk') \in \overline{D}$ and $D \vdash nr' \leq nk'$. Applying (3.10) to this gives two cases:

SubCase: $D \vdash nk'$ empty Because $D \vdash nr' \leq nk'$, Theorem 3.32 (Empty Transitivity) on page 200 gives $D \vdash nr'$ empty, which implies (3.11).

SubCase: Otherwise Then there is a np' such that $\&npcs \succeq c(np') \in \overline{D}$ and $D \vdash nk' \leq np'$. The definition of Q then gives $(nr', np') \in Q$.

End SubCase

Summarizing, (3.11) is true regardless. By NEW-INFER-RECSUB, this implies our conclusion.

Case: $nr \propto \&\{r_i \to nr_i \mid i \in 1 \ldots n\}$ Then the last inference of $D \vdash nr \leq nk$ must be ARROW-RECSUB, so $nk \propto \{k_j \to nk_j \mid j \in 1 \ldots m\}$ and, similarly, $np \propto \{p_z \to np_z \mid z \in 1 \ldots q\}$. The premises of ARROW-RECSUB must include

$$D \vdash \ \&\{r_i \to nr_i \mid i \in 1 \ldots n\} \sqsubset t \tag{3.12}$$

$$\text{for } j \in 1 \ldots m \text{ we have } D \vdash \&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \leq nk_j \tag{3.13}$$

$$\text{for } z \in 1 \ldots q \text{ we have } D \vdash \&\{nk_j \mid j \in 1 \ldots m \text{ and } p_z \leq k_j\} \leq np_z \tag{3.14}$$

By the form of nr, we must have $t \propto t_1 \to t_2$.

By repeated use of Lemma 3.29 (Recursive Intersection Lower Bound) on page 196 with (3.13) we have

$$\begin{array}{l}\text{for } z \in 1 \ldots q \text{ and } j' \in 1 \ldots m \text{ we have} \\ \quad p_z \leq k_{j'} \text{ implies} \\ \quad\quad D \vdash \&\{\&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \mid j \in 1 \ldots m \text{ and } p_z \leq k_j\} \leq nk_{j'}\end{array}$$

and Fact 3.30 (Recursive Intersection Greatest) on page 198 then gives

$$\begin{array}{l}\text{for } z \in 1 \ldots q \text{ we have} \\ \quad D \vdash \&\{\&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \mid j \in 1 \ldots m \text{ and } p_z \leq k_j\} \leq \\ \quad\quad \&\{nk_{j'} \mid j' \in 1 \ldots m \text{ and } p_z \leq k_{j'}\}.\end{array}$$

Set manipulation gives

$$\{\&\{nr_i \mid i \in 1 \ldots n \text{ and } k_j \leq r_i\} \mid j \in 1 \ldots m \text{ and } p_z \leq k_j\}$$
$$=$$
$$\{nr_i \mid i \in 1 \ldots n \text{ and } j \in 1 \ldots m \text{ and } p_z \leq k_j \text{ and } k_j \leq r_i\}.$$

By TRANS-SUBTYPE, $p_z \leq k_j$ and $k_j \leq r_i$ implies $p_z \leq r_i$, so

$$\{nr_i \mid i \in 1 \ldots n \text{ and } j \in 1 \ldots m \text{ and } p_z \leq k_j \text{ and } k_j \leq r_i\}$$
$$\subset$$
$$\{nr_i \mid i \in 1 \ldots n \text{ and } p_z \leq r_i\}.$$

Thus Lemma 3.29 (Recursive Intersection Lower Bound) on page 196 gives

for $z \in 1 \ldots q$ we have
$D \vdash \&\{nr_i \mid i \in 1 \ldots n \text{ and } p_z \leq r_i\} \leq \&\{nk_{j'} \mid j' \in 1 \ldots m \text{ and } p_z \leq k_{j'}\}$

From this, (3.14), and the definition of Q, we can infer

$$\text{for } z \in 1 \ldots q \text{ we have } (\&\{nr_i \mid i \in 1 \ldots n \text{ and } p_z \leq r_i\}, np_z) \in Q,$$

and ARROW-RECSUB then gives $(nr, np) \in F(Q)$, which is our conclusion.

Case: $nr \propto \&rcs$ Then the last inference of $D \vdash nr \leq nk$ is OLD-RECSUB and $nk \propto \&kcs$. Similarly, $np \propto \&pcs$. The premises of OLD-RECSUB are $(\overset{\text{def}}{\wedge} rcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge} kcs)$ and $(\overset{\text{def}}{\wedge} kcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge} pcs)$. By Assumption 2.14 (trans-$\overset{\text{def}}{\leq}$) on page 34 we have $(\overset{\text{def}}{\wedge} rcs) \overset{\text{def}}{\leq} (\overset{\text{def}}{\wedge} pcs)$, and then OLD-RECSUB gives $(nr, np) \in F(Q)$, which is our conclusion.

Case: $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ Then the last inference of $D \vdash nr \leq nk$ is TUPLE-RECSUB, so $nk \propto (nk_{11} * \ldots * nk_{1n}) \,\&\, \ldots \,\&\, (nk_{q1} * \ldots * nk_{qn})$. Similarly, $np \propto (np_{11} * \ldots * np_{1n}) \,\&\, \ldots \,\&\, (np_{z1} * \ldots * np_{zn})$ and the premises of TUPLE-RECSUB are

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \leq nk_{1i} \,\&\, \ldots \,\&\, nk_{qi}$$

and

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash nk_{1i} \,\&\, \ldots \,\&\, nk_{qi} \leq np_{1i} \,\&\, \ldots \,\&\, np_{zi}.$$

The definition of Q then gives

$$\text{for } i \in 1 \ldots n \text{ we have } (nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}, np_{1i} \,\&\, \ldots \,\&\, np_{zi}) \in Q,$$

and TUPLE-RECSUB then gives $(nr, np) \in F(Q)$, which is our conclusion. □

Finally, we can show that recursive subtyping is sound:

Theorem 3.34 (Recursive Subtype Soundness) *If $D \vdash nr \leq nk$ and $D \vdash v \in nr$ then $D \vdash v \in nk$.*

Proof: By co-induction. Take D to be fixed, and let F be the natural encoding of the recursive subtyping relation defined in Figure 3.6 as a function from pairs of recursive types to pairs of recursive types, and let

$$Q = \{(v, nk) \mid \text{for some } nr \text{ we have } D \vdash nr \leq nk \text{ and } D \vdash v \in nr\}.$$

We want to show $Q \subset \text{gfp}(F)$, and by co-induction it suffices to show $Q \subset F(Q)$. Proof is by cases on a $(v, nk) \in Q$. The proof is straightforward, but we include it because the result is important.

Case: $nk \propto \&nks$ where nks has two or more elements Then there is an nr such that $D \vdash nr \leq \&nks$ and $D \vdash v \in nr$. By Lemma 3.31 (Self Recsub) on page 199,

$$\text{for all } nk' \in nks \text{ we have } D \vdash \&nks \leq nk'$$

and by Theorem 3.33 (Subtype Transitivity) on page 201,

$$\text{for all } nk' \in nks \text{ we have } D \vdash nr \leq nk'.$$

By definition of Q, this implies

$$\text{for all } nk' \in nks \text{ we have } (v, nk') \in Q$$

and by AND-RECVALUE this implies $(v, \&nks) \in F(Q)$, which is our conclusion.

Case: $nk \propto k \to nk'$ Then there is an nr such that $D \vdash nr \leq k \to nk'$ and $D \vdash v \in nr$. The only way to infer the first of these is by using ARROW-RECSUB, so $nr \propto \&\{r_i \to nr_i \mid i \in 1 \ldots n\}$ and the premises of ARROW-RECSUB are

$$D \vdash \ \&\ \{r_i \to nr_i \mid i \in 1 \ldots n\} \sqsubset t$$

and

$$D \vdash \&\{nr_i \mid i \in 1 \ldots n \text{ and } k \leq r_i\} \leq nk'. \tag{3.15}$$

The last inferences of $D \vdash v \in nr$ must be AND-RECVALUE and ABS-RECVALUE, so $v \propto$ `fn` $x : u$ `=>` e and the premises of ABS-RECVALUE are

$$\begin{array}{l} \text{for } i \text{ in } 1 \ldots n, \text{ all } v', \text{ and all } v'' \text{ such that} \\ \quad \cdot \vdash v'' : k \text{ and} \\ \quad (\texttt{fn}\ x : u\ \texttt{=>}\ e)\ v'' \Rightarrow v' \\ \text{we have} \\ \quad D \vdash v' \in nr_i \end{array} \tag{3.16}$$

We want to show $((\texttt{fn}\ x{:}t\ \texttt{=>}\ e), k \to nk') \in F(Q)$. We can only infer this by using ABS-RECVALUE with the premise

$$\begin{array}{l}\text{for all } v' \text{ and } v'' \text{ such that}\\ \quad \cdot \vdash v'' : k \text{ and}\\ \quad (\texttt{fn}\ x{:}t\ \texttt{=>}\ e)\ v'' \Rightarrow v'\\ \text{we have}\\ \quad (v', nk') \in Q\end{array} \tag{3.17}$$

To prove this, let v' and v'' be given such that $\cdot \vdash v'' : k$ and $(\texttt{fn}\ x{:}t\ \texttt{=>}\ e)\ v'' \Rightarrow v'$. By (3.16),

$$\text{if } i \in 1 \ldots n \text{ and } k \leq r_i \text{ then } D \vdash v' \in nr_i,$$

and then AND-RECVALUE gives $D \vdash v' \in \&\{nr_i \mid i \in 1 \ldots n \text{ and } k \leq r_i\}$. Then (3.15) and the definition of Q imply $(v', nk') \in Q$. Thus (3.17) is true, which implies our conclusion.

Case: $nk \propto nkc$ The last inference in $D \vdash nr \leq nk$ must be NEW-INFER-RECSUB where $nr \propto \&nrcs$ and the premises of NEW-INFER-RECSUB are

$$\frac{D \vdash \&\, nrcs \sqsubset tc \qquad D \vdash nkc \sqsubset tc}{}$$

$$\begin{array}{l}\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D}\\ \text{either } D \vdash nr' \text{ empty}\\ \text{or there is a } nk' \text{ such that } nkc \succeq c(nk') \in D \text{ and } D \vdash nr' \leq nk'.\end{array} \tag{3.18}$$

The last inferences of $D \vdash v \in nr$ must be AND-RECVALUE and NEW-RC-RECVALUE where $v \propto c\ v'$ and the premises of NEW-RC-RECVALUE are

$$\begin{array}{l}\text{for all } nrc \in nrcs \text{ we have some } nr'_{nrc} \text{ such that}\\ nrc \succeq c(nr'_{nrc}) \in D \text{ and}\\ D \vdash v' \in nr'_{nrc}\end{array}$$

Let $np = \&\{nr'_{nrc} \mid nrc \in nrcs\}$. By definition of intersection membership, $\&nrcs \succeq c(np) \in \overline{D}$. By AND-RECVALUE, $D \vdash v' \in np$. By Theorem 3.21 (Soundness of Empty) on page 191, we cannot have $D \vdash np$ empty, so (3.18) implies there is a nk' such that $nkc \succeq c(nk') \in D$ and $D \vdash np \leq nk'$. By definition of Q, this implies $(v', nk') \in Q$, and NEW-RC-RECVALUE then gives $(c\ v', nkc) \in F(Q)$, which is what we wanted to show.

Case: $nk \propto kc$ Then the last inference of $D \vdash nr \leq nk$ is OLD-RECSUB where $nr \propto \&rcs$ and the premise of OLD-RECSUB is $(\overset{\text{def}}{\wedge}\ rcs) \overset{\text{def}}{\leq} kc$. The last inferences of $D \vdash v \in nr$ must be OLD-RC-RECVALUE and AND-RECVALUE with the premises

$$\text{for } rc \in rcs \text{ we have } \cdot \vdash v : rc.$$

Simple manipulation of refinement types then gives $\cdot \vdash v : \overset{\text{def}}{\wedge} rcs$; then WEAKEN-TYPE gives $\cdot \vdash v : kc$ and OLD-RC-RECVALUE gives $(v, kc) \in F(Q)$, which is our conclusion.

Case: $nk \propto nk_1 * \ldots * nk_n$ Then the last inference of $D \vdash nr \leq nk$ is TUPLE-RECSUB and $nr \propto (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn})$ and the premise of TUPLE-RECSUB is

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \leq nk_i.$$

The last inferences of $D \vdash v \in nk$ must be AND-RECVALUE and TUPLE-RECVALUE where $v \propto (v_1, \ldots, v_n)$ and the premises of TUPLE-RECVALUE are

$$\text{for } i \in 1 \ldots n \text{ and } j \in 1 \ldots m \text{ we have } D \vdash v_i \in nr_{ji}.$$

AND-RECVALUE gives

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash v_i \in nr_{1i} \,\&\, \ldots \,\&\, nr_{mi}.$$

Then the definition of Q gives

$$\text{for } i \in 1 \ldots n \text{ we have } (v_i, nk_i) \in Q$$

and TUPLE-RECVALUE then gives $((v_1, \ldots, v_n), nk_1 * \ldots * nk_n) \in F(Q)$, which is our conclusion. □

3.5 Splitting

This section describes a type inference system for inferring the relation $\overset{\text{def}}{\asymp}$ from an abstract declaration.

The type inference rules for splitting recursive types are in Figure 3.8. The $\times$ operator used in the TUPLE-RECSPLIT rule was introduced on page 117. The ARROW-RECSPLIT and TUPLE-RECSPLT rules are straightforward; we shall explain the other two.

The main idea behind the NEW-RECSPLIT rule is, wherever the type $\&nrcs$ has the definition $c(nr)$, there must be some consistency between the splits of nr and the splits of $\&nrcs$. One way to understand the details is by stepping through an explanation of why it is sound. Suppose a value $c\ v$ is in $\&nrcs$; then we need to know it is in some fragment $\&nkcs$ of $\&nrcs$. There must be some definition $c(nr)$ of $\&nrcs$ such that v is in nr. Thus v is in some fragment nk of nr. If there is some definition $c(np)$ of $\&nkcs$ such that np is larger than nk, then we know that v is in np and $c\ v$ is in $\&nkcs$.

Another way to understand it is to look at an example. If we take the `datatype` declaration

$$\texttt{datatype}\ blist\ \texttt{=}\ \texttt{cons of}\ bool * blist\ \texttt{|}\ \texttt{nil of}\ runit$$

NEW-RECSPLIT:
$$\frac{\begin{array}{l} s \text{ is a nonempty set, and} \\ \text{for all } nk \text{ in } s \text{ we have } D \vdash nk \leq \&nrcs, \text{ and} \\ \text{for all } c \text{ and } nr \text{ such that } \&nrcs \succeq c(nr) \in \overline{D} \\ \text{there is an } s' \text{ such that} \\ \quad D \vdash nr \asymp s' \\ \quad \text{for all } nk \in s', \text{ there is an } \&nkcs \in s \text{ and a } np \text{ such that} \\ \qquad \&nkcs \succeq c(np) \in \overline{D} \\ \qquad D \vdash nk \leq np \end{array}}{D \vdash \ \& \ nrcs \asymp s}$$

ARROW-RECSPLIT:
$$\frac{}{D \vdash r_1 \to nr_1 \,\&\, \ldots \,\&\, r_n \to nr_n \asymp \{r_1 \to nr_1 \,\&\, \ldots \,\&\, r_n \to nr_n\}}$$

OLD-RECSPLIT:
$$\frac{rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \overset{\text{def}}{\asymp} \{\overset{\text{def}}{\wedge} rcs \mid rcs \in s\}}{D \vdash nr_1 \,\&\, \ldots \,\&\, nr_n \asymp \{\&rcs \mid rcs \in s\}}$$

TUPLE-RECSPLIT:
$$\frac{\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i} \,\&\, \ldots \,\&\, nr_{mi} \asymp s_i}{D \vdash (nr_{11} * \ldots * nr_{1n}) \,\&\, \ldots \,\&\, (nr_{m1} * \ldots * nr_{mn}) \asymp s_1 \times \ldots \times s_n}$$

Figure 3.8: Splitting for Recursive Types (Greatest Fixed Point)

and the abstract declaration

$$\begin{array}{rl} D = \{ & \top_{blist} \succeq \texttt{cons}(\top_{bool} * \top_{blist}) \\ & \top_{blist} \succeq \texttt{nil}(runit) \\ & bev \succeq \texttt{cons}(\top_{bool} * bod) \\ & bev \succeq \texttt{nil}(runit) \\ & bod \succeq \texttt{cons}(\top_{bool} * bev)\}, \end{array}$$

we can infer $D \vdash \top_{blist} \asymp \{bev, bod\}$. The root inference of the derivation of this is NEW-RECSPLIT with the premises:

$$\begin{array}{c} \{bev, bod\} \text{ is a nonempty set} \\ D \vdash bev \leq \top_{blist} \\ D \vdash bod \leq \top_{blist} \\ D \vdash \top_{bool} * \top_{blist} \asymp \{\top_{bool} * bev, \top_{bool} * bod\} \\ bev \succeq \texttt{cons}(\top_{bool} * bod) \in D \\ D \vdash \top_{bool} * bod \leq \top_{bool} * bod \\ bod \succeq \texttt{cons}(\top_{bool} * bev) \in D \\ D \vdash \top_{bool} * bev \leq \top_{bool} * bev \\ D \vdash runit \asymp \{runit\} \\ bev \succeq \texttt{nil}(runit) \in D \\ D \vdash runit \leq runit \end{array}$$

Despite the lack of intuitiveness of this rule, it seems to work well in practice.

A previous version of OLD-RECSPLIT simply said

$$\frac{rc_1 \overset{\text{def}}{\wedge} \ldots \overset{\text{def}}{\wedge} rc_n \overset{\text{def}}{\asymp} s}{D \vdash nr_1 \,\&\, \ldots \,\&\, nr_n \asymp s}$$

which is straightforward. The problem with this definition is that we need to prove an analogue of Lemma 2.43 (Split Intersection) on page 54 for recursive types. This requires sometimes having intersections of recursive types as fragments of recursive types; the previous version does not permit this, but the rule as stated allows it.

It is not clear how to make an efficient algorithm for this inference system. The implementation attempts to use a brute-force search to find the principal splits directly; I do not know if that strategy is sound. Roughly speaking, the implementation enumerates all fixed points of the function arising from this inference system such that each intersection of recursive type constructors has exactly one split, and that no two elements of that split are comparable. A fixed point that contains the smallest types in the splits is chosen, and we assume it contains principal splits. I do not know whether there will always be a fixed point with the least types.

We can show if a value is in a recursive type, then it is in some fragment of that recursive type.

Theorem 3.35 (Recursive Split Soundness) *If $D \vdash nr \asymp nks$ and $D \vdash v \in nr$ then for some $nk \in nks$ we have $D \vdash v \in nk$.*

Proof: By induction on v.

Case: $v \propto c\ \ v'$ where c is new | Then the last inferences of $D \vdash v \in nr$ must be AND-RECVALUE and NEW-RC-RECVALUE so nr has the form $\&nrcs$ and the premises of NEW-RC-RECVALUE are

$$\text{for } nrc \in nrcs \text{ we have } nrc \succeq c(nr_{nrc}) \in D \tag{3.19}$$

and

$$\text{for } nrc \in nrcs \text{ we have } D \vdash v' \in nr_{nrc}. \tag{3.20}$$

The last inference of $D \vdash nr \asymp nks$ must be NEW-RECSPLIT with the premises

$$nks \text{ is a nonempty set}$$

$$\text{for all } nk \in kts \text{ we have } D \vdash nk \leq \&nrcs$$

and

$$\begin{array}{l}\text{for all } c \text{ and all } nr' \text{ such that } \&nrcs \succeq c(nr') \in \overline{D} \\ \text{there is an } s' \text{ such that} \\ \quad D \vdash nr' \asymp s' \\ \quad \text{for all } np' \in s' \text{ there is an } \&nkcs \in nks \text{ and a } np \text{ such that} \\ \quad\quad \&nkcs \succeq c(np) \in \overline{D} \\ \quad\quad D \vdash np' \leq np\end{array} \tag{3.21}$$

Let $nr' = \&\{nr_{nrc} \mid nrc \in nrcs\}$. Using the definition of intersection membership and (3.19) gives $\&nrcs \succeq c(nr') \in \overline{D}$. Thus we can use (3.21) to get an s' such that

$$D \vdash nr' \asymp s'$$

and

$$\begin{array}{l}\text{for all } np' \in s' \text{ there is an } \&nkcs \in nks \text{ and a } np \text{ such that} \\ \quad \&nkcs \succeq c(np) \in \overline{D} \\ \quad D \vdash np' \leq np.\end{array} \tag{3.22}$$

Using AND-RECVALUE on (3.20) gives $D \vdash v' \in nr'$, so our induction hypothesis gives a $np' \in s'$ such that $D \vdash v' \in np'$. Thus (3.22) gives a $\&nkcs \in nks$ and a np such that $\&nkcs \succeq c(np) \in \overline{D}$ and $D \vdash np' \leq np$. Theorem 3.34 (Recursive Subtype Soundness) on page 204 gives $D \vdash v' \in np$, and Fact 3.11 (Intersection Value Membership) on page 183 then gives $D \vdash c\ \ v' \in \&nkcs$, which is our conclusion.

The remaining cases are simple, but they are also short, so we include them for completeness.

Case: $v \propto \texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ e$ Then the last inferences of $D \vdash v \in nr$ must be AND-RECVALUE and ABS-RECVALUE where $nr \propto r_1 \to nr_1\ \&\ \ldots\ \&\ r_n \to nr_n$. Thus the only way to infer $D \vdash nr \asymp nks$ is by using ARROW-RECSPLIT so $nks = \{nr\}$ and our premise $D \vdash nr \asymp nks$ is our conclusion.

Case: $v \propto c\ \ v'$ where c is old Then the last inferences of $D \vdash v \in nr$ are AND-RECVALUE and OLD-RC-RECVALUE where $nr \propto \&rcs$ and the premises of OLD-RC-RECVALUE are

$$\text{for } nr \in rcs \text{ we have } \cdot \vdash c\ \ v' : rc.$$

The last inference of $D \vdash nr \asymp nks$ must be OLD-RECSPLIT where for some set s of sets of refinement type constructors, $kts = \{\&kcs \mid kcs \in s\}$ and the premise of OLD-RECSPLIT is $\overset{\text{def}}{\wedge} rcs \overset{\text{def}}{\asymp} \{\overset{\text{def}}{\wedge} kcs \mid kcs \in s\}$. Simple reasoning about refinement types gives $\cdot \vdash c\ \ v' : \&rcs$ and Theorem 2.69 (Splitting Value Types) on page 89 gives a $\&kcs \in nks$ such that $\cdot \vdash c\ \ v' : \overset{\text{def}}{\wedge} kcs$. By WEAKEN-TYPE, OLD-RC-RECVALUE, and AND-RECVALUE this implies $D \vdash c\ \ v' \in \&kcs$, which is our conclusion.

Case: $v \propto (v_1,\ \ldots,\ v_n)$ Then the last inferences of $D \vdash v \in nr$ must me AND-RECVALUE and TUPLE-RECVALUE where $nr \propto (nr_{11} * \ldots * nr_{1n}) \& \ldots \& (nr_{m1} * \ldots * nr_{mn})$ and the premises are

$$\text{for } i \in 1 \ldots n \text{ and } j \in 1 \ldots m \text{ we have } D \vdash v_i \in nr_{ji}.$$

The last inference of $D \vdash nr \asymp nks$ must be TUPLE-RECSPLIT where $nks \propto s_1 \times \ldots \times s_n$ and the premises of TUPLE-RECSPLIT are

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash nr_{1i}\ \&\ \ldots\ nr_{mi} \asymp s_i.$$

AND-RECVALUE gives

$$\text{for } i \in 1 \ldots n \text{ we have } D \vdash v_i \in nr_{1i} \,\&\, \ldots \,\&\, nr_{mi},$$

and then our induction hypothesis gives

$$\text{for } i \in 1 \ldots n \text{ there is a } nk_i \in s_i \text{ such that } D \vdash v_i \in nk_i.$$

TUPLE-RECVALUE then gives $D \vdash (v_1,\ \ldots,\ v_n) \in nk_1 * \ldots * nk_n$. The definition of $\times$ gives $nk_1 * \ldots * nk_n \in nks$, which is our conclusion. □

It is also possible to show that intersection interacts with splitting in a natural way. Compare this to Lemma 2.43 (Split Intersection) on page 54.

Fact 3.36 (Recursive Split Intersection) *If $D \vdash nr \asymp s$ and $D \vdash nr \sqsubset t$ and $D \vdash nk \sqsubset t$, then $D \vdash nr \,\&\, nk \asymp \{np \,\&\, nk \mid np \in s\}$.*

Proof of this is a straightforward co-induction.

3.6 Recursive Types provide Refinement Type Constructors

In this section we show that the assumptions made in Chapter 2 and the assumptions made about $\overset{\text{def}}{\text{empty}}$ in this chapter actually hold for recursive types as defined in this chapter.

In Subsection 3.6.1, we define the operators that were taken as predefined in Chapter 2 in terms of recursive type operations defined in this chapter. Then in Subsection 3.6.2 we enumerate the assumptions from Chapter 2 and prove them. Finally, in Subsection 3.6.3 we will prove a grand soundness result for this entire chapter: if refinement type inference concludes a value is in a refinement type, then it is also in the corresponding recursive type.

3.6.1 Defining the Primitives

First we need to define the primitives $\overset{\text{def}}{\wedge}$, $\overset{\text{def}}{\leq}$, $\overset{\text{def}}{:}$, and so forth in terms of recursive type inference as defined in this chapter. We assume at this point that there is some specific abstract declaration D we are adding to the global environment, and the primitives are being expanded to include the new declaration. For example, we expect $c \overset{\text{def}}{:} r \hookrightarrow nr$ to be true if either it was true before we encountered the declaration D, or c, r and nr satisfy the definition we give below.

If two refinement type constructors refine the same ML type constructor, then by Assumption 2.16 ($\overset{\text{def}}{\wedge}$ defined) on page 34 their intersection (using $\overset{\text{def}}{\wedge}$) is a refinement type

constructor. This is not true for recursive type constructors, so we cannot simply define the new refinement type constructors to be the new recursive type constructors. Instead, we define a refinement type constructor to be any intersection (using &) of recursive type constructors that all refine the same ML type constructor.

We can easily promote this way to construct refinement type constructors from recursive type constructors to a way to construct refinement types from recursive types. However, since we consider & for recursive types to be associative, commutative, and idempotent, but we do not assume the same for $\wedge$ for refinement types, there are many refinement types corresponding to one recursive type. For example, the recursive type bem & bod & $\top_{blist}$ corresponds to any of the refinement types

$$\begin{gathered} bem \wedge bod \wedge \top_{blist}, \\ bod \wedge bem \wedge (\top_{blist} \,\&\, bem), \\ bem \,\&\, bod \,\&\, \top_{blist}, \end{gathered}$$

or infinitely many others. We could represent this as a one-to-many relation between recursive types and refinement types. Instead, we will represent it as the inverse of a many-to-one function rtort from *r*efinement types *to r*ecursive *t*ypes. (Hence the name rtort.) Formally, we have the following definition:

Definition 3.37 *Define the function* rtort *from refinement types to recursive types by the recursion equations*

$$\begin{aligned} \text{rtort}(r_1 \to r_2) &= r_1 \to \text{rtort}(r_2) \\ \text{rtort}(r_1 \wedge r_2) &= \text{rtort}(r_1) \,\&\, \text{rtort}(r_2) \\ \text{rtort}(rc) &= rc \\ \text{rtort}(\&nrcs) &= \&nrcs \\ \text{rtort}(r_1 * \ldots * r_n) &= \text{rtort}(r_1) * \ldots * \text{rtort}(r_n) \end{aligned}$$

It is very straightforward to define when a refinement type constructor refines an ML type constructor in terms of the behavior of the recursive types:

Definition 3.38 *We say* $\&nrcs \stackrel{\text{def}}{\sqsubset} tc$ *if* $D \vdash \,\&\, nrcs \sqsubset tc$.

With this definition, recursive types refine ML types if and only if the corresponding refinement types refine the same ML types:

Fact 3.39 *Under the assumptions arising from* D *we have* $r \sqsubset t$ *if and only if* $D \vdash \text{rtort}(r) \sqsubset t$.

Proof of this is by induction on r.

To find the intersection of constructed refinement type constructors, we take the intersection at the recursive type level:

Definition 3.40 *We say &nrcs* $\overset{\text{def}}{\wedge}$ *&nkcs* $= \&(nrcs \cup nkcs)$ *whenever there is a* t *such that* $D \vdash \ \&\ (nrcs \cup nkcs) \sqsubset t$.

The definition of subtyping refinement type constructors in terms of subtyping recursive types is also straightforward:

Definition 3.41 *We say &nrcs* $\overset{\text{def}}{\leq}$ *&nkcs if* $D \vdash$ *&nrcs* $\leq$ *&nkcs.*

With this definition, the subtyping relation for refinement types coincides with the one for recursive types:

Fact 3.42 (Refinement and Recursive Subtyping Equivalence) *We have* $D \vdash \text{rtort}(r) \leq \text{rtort}(k)$ *if and only if, under the assumptions arising from* D*, we have* $r \leq k$.

One proof of this takes the "if" and the "only if" cases separately. To prove the "if" case, use Theorem 2.21 (Subtypes Refine) on page 36 to find a t that both r and k refine, and proceed by induction on that t. To prove the "only if" case, we use Fact 3.28 (Recursive Subtypes Refine) on page 196 to find a t that both $\text{rtort}(r)$ and $\text{rtort}(k)$ refine, and again proceed by induction on that t.

This implies that, whenever two refinement types coerce to the same recursive type, they are equivalent:

Corollary 3.43 (Equivalence rtort**)** *If* $\text{rtort}(r) = \text{rtort}(k)$ *and* $r \sqsubset t$ *and* $k \sqsubset t$ *then* $r \equiv k$.

Proof: By Lemma 3.31 (Self Recsub) on page 199, $D \vdash \text{rtort}(r) \leq \text{rtort}(k)$, and Fact 3.42 (Refinement and Recursive Subtyping Equivalence) on page 212 gives $r \leq k$. Similarly $k \leq r$, and together these imply $r \equiv k$. □

This can be used to show that recursive splitting and refinement type splitting are consistent:

Fact 3.44 (Refinement and Recursive Split Consistency) *If* $D \vdash \text{rtort}(r) \asymp \{\text{rtort}(k) \mid k \in s\}$ *and* $r \sqsubset t$ *then* $r \asymp s$.

The proof of this is a straightforward induction on t.

Whenever we have $\&nrcs \succeq c(\text{rtort}(r)) \in \overline{D}$, we can say that $c \overset{\text{def}}{:} r \hookrightarrow \&nrcs$. In general, the converse does not hold, because Assumption 2.52 (Constructor Argument Strengthen) on page 67 and Assumption 2.53 (Constructor Result Weaken) on page 67 place constraints on the behavior of $\overset{\text{def}}{:}$ that may not be satisfied by our abstract declaration. For

example, in the presence of the example abstract declaration appearing on page 183, these constraints give

$$\texttt{cons} \overset{\text{def}}{:} (\top_{bool} * (bev \ \& \ bod)) \hookrightarrow bev$$

because $bev \succeq \texttt{cons}(\top_{bool} * bod) \in D$ and $(bev \ \& \ bod) \overset{\text{def}}{\leq} bod$. Reasoning about $\overset{\text{def}}{:}$ requires first defining a version of intersection membership that allows the argument to the constructor to be strengthened and the result to be weakened:

Definition 3.45 (Weakened Intersection Membership) *We define $\widehat{D}$ to contain all elements of the form &nrcs $> c(nr)$ where there are nk and nkcs such that &nkcs $\succeq c(nk) \in \overline{D}$ and $D \vdash nr \leq nk$ and $D \vdash$ &nkcs $\leq$ &nrcs.*

This relation makes a natural statement about membership of values in recursive types:

Fact 3.46 (Weakened Intersection Soundness) *If &nrcs $> c(nr) \in \widehat{D}$ and $D \vdash v' \in nr$ then $D \vdash c \ v' \in$ &nrcs.*

The proof is little more than two uses of Theorem 3.34 (Recursive Subtype Soundness) on page 204.

We can sometimes use the definition of recursive subtyping to eliminate one of the subtyping assertions in the definition of Weakened Intersection Membership:

Lemma 3.47 (Weakened Intersection Simplification I) *If nrc $> c(nr)$ then either $D \vdash$ nr* empty *or there is a nk such that $D \vdash nr \leq nk$ and nrc $\succeq c(nk) \in D$.*

Proof: The definition of weakened intersection gives $npcs$ and np such that

$$\& \ npcs \succeq c(np) \in \overline{D} \tag{3.23}$$

$$D \vdash nr \leq np \tag{3.24}$$

$$D \vdash \& npcs \leq nrc \tag{3.25}$$

The last inference of (3.25) must be NEW-INFER-RECSUB with the premise

$$\begin{array}{l}\text{for all } c \text{ and all } np' \text{ such that } \& npcs \succeq c(np') \in \overline{D} \\ \text{either } D \vdash np' \text{ empty} \\ \text{or there is a } nk \text{ such that } nrc \succeq c(nk) \in D \text{ and } D \vdash np' \leq nk\end{array}.$$

Using (3.23) and (3.25), we get

$$\begin{array}{l}\text{either } D \vdash np \text{ empty} \\ \text{or there is a } nk \text{ such that } nrc \succeq c(nk) \in D \text{ and } D \vdash np \leq nk\end{array}. \tag{3.26}$$

If $D \vdash np$ empty, then Theorem 3.32 (Empty Transitivity) on page 200 and (3.24) give $D \vdash nr$ empty, which implies our conclusion.

Otherwise, let nk be as given in (3.26). Thus $nrc \succeq c(nk) \in D$ and $D \vdash np \leq nk$; Theorem 3.33 (Subtype Transitivity) on page 201 and (3.24) give $D \vdash rc \leq nk$. These imply our conclusion. □

We can also do the same simplification if we replace nrc by an intersection of recursive type constructors:

Lemma 3.48 (Weakened Intersection Simplification II) *If &nrcs $> c(nr) \in \overline{D}$ then either $D \vdash$ nr* empty *or there is a nk such that $D \vdash$ nr $\leq$ nk and &nrcs $\succeq c(nk) \in \overline{D}$.*

Proof: By Lemma 3.31 (Self Recsub) on page 199,

$$\text{for all } nrc \in nrcs \text{ we have } D \vdash \&nrcs \leq nrc.$$

By definition of weakened intersection, $\&nrcs > c(nr) \in \widehat{D}$ implies there are $\&nkcs$ and nk such that

$$\&nkcs \succeq c(nk) \in \overline{D},$$
$$D \vdash \&nkcs \leq \&nrcs,$$

and

$$D \vdash nr \leq nk.$$

Theorem 3.33 (Subtype Transitivity) on page 201 gives

$$\text{for all } nrc \in nrcs \text{ we have } D \vdash \&nkcs \leq nrc$$

and definition of weakened intersection then gives

$$\text{for all } nrc \in nrcs \text{ we have } nrc > c(nr) \in \widehat{D}.$$

and then Lemma 3.47 (Weakened Intersection Simplification I) on page 213 gives

$$\begin{array}{l}\text{for all } nrc \in nrcs\text{, either} \\ \quad D \vdash nr \text{ empty} \\ \text{or there is a } nk_{nrc} \text{ such that} \\ \quad D \vdash nr \leq nk_{nrc} \\ \quad nrc \succeq c(nk_{nrc}) \in D\end{array} \tag{3.27}$$

If $D \vdash nr$ empty, we are done. Otherwise let $nk = \&\{nk_{nrc} \mid nrc \in nrcs\}$. By (3.27) and the definition of weakened intersection, we have $\&nrcs \succeq c(nk) \in \overline{D}$, and by (3.27) and Fact 3.30 (Recursive Intersection Greatest) on page 198 we have $D \vdash nr \leq nk$. These last two are our conclusion. □

We can define $\overset{\text{def}}{:}$ in terms of this. To make Assumption 2.50 (Split Constructor Consistent) on page 66 hold, we need to also say that $c \overset{\text{def}}{:} r \hookrightarrow rc$ whenever r is empty; see the counterexample that arises if we do not assume this in the discussion of Split Constructor Consistent on page 217.

Definition 3.49 *We say* $c \overset{\text{def}}{::} r \hookrightarrow$ *&nrcs if either &nrcs* $> c(\text{rtort}(r)) \in \widehat{D}$*, or all of the following hold:*

$$D \vdash \text{rtort}(r)\ \text{empty}$$

and for some t *and tc we have*

$$c \overset{\text{def}}{::} t \hookrightarrow tc,$$

$$D \vdash\ \&\, nrcs \sqsubset tc,$$

and

$$D \vdash \text{rtort}(r) \sqsubset t.$$

We define $\overset{\text{def}}{\text{empty}}$ in terms of emptyness for recursive types:

Definition 3.50 *We say &nrcs* $\overset{\text{def}}{\text{empty}}$ *if* $D \vdash$ *&nrcs* empty.

Finally, we define splitting for constructed refinement type constructors in terms of splitting for recursive types:

Definition 3.51 *If* $D \vdash\ \&\, nrcs \asymp s$*, then we say &nrcs* $\overset{\text{def}}{\asymp} \{\&nkcs \mid \&nkcs \in s\}$.

With these assumptions, refinement type emptyness and recursive type emptyness coincide:

Fact 3.52 (Emptyness Consistency) *If* $D \vdash \text{rtort}(r)$ empty *then under the assumptions introduced by* D *we have* $\vdash r$ empty.

The proof is by induction on $\text{depth}(\text{rtort}(r))$.

3.6.2 Proving the Assumptions

In this subsection we enumerate the assumptions made in Chapter 2 about predefined properties of refinement type constructors, and prove that they hold for refinement type constructors derived from recursive types as described in this chapter. This is only non-trivial for Split Constructor Consistent, which is discussed on page 217.

Some assumptions are not addressed in this list because they concern only ML type inference, which for the purposes of this thesis we assume is well-understood. We include them in the list with the statement that the assumption is entirely about ML types.

Assumption 2.2 (Constructors have Unique ML Types) on page 26: For each c, there are unique t and tc such that

$$c \overset{\text{def}}{::} t \hookrightarrow tc.$$

This is entirely about ML types.

Assumption 2.7 (Unique Predefined Refinements) on page 31: For all rc there is a unique tc such that $rc \stackrel{\text{def}}{\sqsubset} tc$.

Proof: If rc is old, this follows from Unique Predefined Refinements before we incorporated D into the environment.

Otherwise, rc is new and has the form $\&nrcs$ where there is a tc such that $D \vdash \&nrcs \sqsubset tc$. Thus there is at least one $rc \sqsubset tc$.

To show there is at most one, suppose $D \vdash \ \& \ nrcs \sqsubset tc$ and $D \vdash \ \& \ nrcs \sqsubset tc'$. Let nrc be any element of $nrcs$. By AND-RECREFINES we must have $D \vdash nrc \sqsubset tc$ and $D \vdash nrc \sqsubset tc'$. The last inference of these must be NEW-RECREFINES with the premises

$$\begin{array}{l}\text{for all } c \text{ and } nr \text{ such that } nrc \succeq c(nr) \in D \\ \quad \text{there is a } t \text{ such that } c \stackrel{\text{def}}{::} t \hookrightarrow tc \\ \text{for all } c \text{ and } nr \text{ such that } nrc \succeq c(nr) \in D \\ \quad \text{there is a } t \text{ such that } c \stackrel{\text{def}}{::} t \hookrightarrow tc'\end{array}$$

By Condition 3.2 (New Recursive Type Constructors Defined) on page 178 these universal quantifications are not vacuous, so by Assumption 2.2 (Constructors have Unique ML Types) on page 26, $tc = tc'$. □

Assumption 2.8 (Finite Predefined Refinements) on page 31: Immediate from Condition 3.6 (Declarations are Finite) on page 179.

Assumption 2.13 (reflex-$\stackrel{\text{def}}{\leq}$) on page 33: Immediate from Lemma 3.31 (Self Recsub) on page 199.

Assumption 2.14 (trans-$\stackrel{\text{def}}{\leq}$) on page 34: Immediate from Theorem 3.33 (Subtype Transitivity) on page 201.

Assumption 2.15 (Refines $\stackrel{\text{def}}{\leq}$) on page 34: Immediate from Fact 3.28 (Recursive Subtypes Refine) on page 196 and Fact 3.9 (Recursive Unique ML Types) on page 179.

Assumption 2.16 ($\stackrel{\text{def}}{\wedge}$ defined) on page 34: Immediate from the definition of refinement type constructors.

Assumption 2.17 ($\stackrel{\text{def}}{\wedge}$ Elim) on page 34: Immediate from Lemma 3.31 (Self Recsub) on page 199.

Assumption 2.18 (and-intro-$\stackrel{\text{def}}{\leq}$) on page 34: Immediate from Fact 3.30 (Recursive Intersection Greatest) on page 198.

Assumption 2.30 (Split Subtype Consistent) on page 49: Immediate from the second premise of NEW-RECSPLIT.

Assumption 2.36 (Refinement Constructor Splits are Nonempty) on page 51: Immediate from the first premise of NEW-RECSPLIT.

Assumption 2.42 (Predefined Split Intersection) on page 54: Immediate from Fact 3.36 (Recursive Split Intersection) on page 210.

Assumption 2.49 (Constructor Type Refines) on page 65: Straightforward from Fact 3.28 (Recursive Subtypes Refine) on page 196.

Assumption 2.50 (Split Constructor Consistent) on page 66: If

$$c \stackrel{\text{def}}{:} r \hookrightarrow rc$$

and

$$rc \stackrel{\text{def}}{\asymp} \{rc_1, \ldots, rc_n\}$$

then there is some provable assertion of the form

$$r \asymp \{r_1, \ldots, r_m\}$$

such that for all j between 1 and m there is an i between 1 and n such that

$$c \stackrel{\text{def}}{:} r_j \hookrightarrow rc_i.$$

If instead we defined $c \stackrel{\text{def}}{:} r \hookrightarrow \&nrcs$ to mean $\&nrcs > c(\text{rtort}(r)) \in \widehat{D}$, this would not be true. A counterexample presumes the datatype declaration

$$\texttt{datatype}\ d\ \texttt{=}\ \texttt{a of}\ tunit\ \texttt{|}\ \texttt{b of}\ tunit\ \texttt{|}\ \texttt{c of}\ d$$

and the abstract declaration

$$\begin{aligned} D = \{&em \succeq \text{bottom}(d),\\ &nem1 \succeq \texttt{a}(runit),\\ &nem1 \succeq \texttt{b}(runit),\\ &nem2 \succeq \texttt{a}(runit),\\ &nem2 \succeq \texttt{c}(em),\\ &nem3 \succeq \texttt{a}(runit),\\ &nem4 \succeq \texttt{b}(runit)\}. \end{aligned}$$

The names em and nem stand for "empty" and "nonempty", respectively. Because $D \vdash nem1 \leq nem2$, we have $nem1 > \texttt{c}(em) \in \widehat{D}$. We also have $D \vdash nem1 \asymp \{nem3, nem4\}$, so Split Constructor Consistent would give a split of em where c maps each fragment to either $em3$ or $em4$. Since the fragments must all be subtypes of em, the only possible split is $\{em\}$. If we had the simpler definition of $\stackrel{\text{def}}{:}$, this would imply that either $nem3 > \texttt{c}(em) \in \widehat{D}$ or $nem4 > \texttt{c}(em) \in \widehat{D}$, neither of which is true. With the actual definition of $\stackrel{\text{def}}{:}$, we have both $\texttt{c} \stackrel{\text{def}}{:} em \hookrightarrow nem3$ and $\texttt{c} \stackrel{\text{def}}{:} em \hookrightarrow nem4$. In fact, we can prove that Split Constructor Consistent is true in general for the actual definition of $\stackrel{\text{def}}{:}$.

Proof: We must have $nr \propto \&nrcs$. By definition of $\overset{\text{def}}{:}$, either $D \vdash \text{rtort}(r)$ empty or $\&nrcs > c(\text{rtort}(r)) \in \widehat{D}$. In the latter case, Lemma 3.48 (Weakened Intersection Simplification II) on page 214 gives

$$\begin{array}{l}\text{either } D \vdash \text{rtort}(r) \text{ empty} \\ \text{or there is a } nk \text{ such that} \\ \quad D \vdash \text{rtort}(r) \leq nk \\ \quad \&nrcs \succeq c(nk) \in \overline{D}.\end{array}$$

If $D \vdash \text{rtort}(r)$ empty, by SELF-SPLIT we can choose $s = \{r\}$. Let kc be any element of sc; then the definition of $\overset{\text{def}}{:}$ immediately gives $c \overset{\text{def}}{:} r \hookrightarrow kc$, which is our conclusion.

Otherwise, there is a nk such that $D \vdash \text{rtort}(r) \leq nk$ and $\&nrcs \succeq c(nk) \in \overline{D}$. If we let $sc' = \{\text{rtort}(p) \mid p \in sc\}$ the definition of $\overset{\text{def}}{\asymp}$ gives $D \vdash \ \& \ nrcs \asymp sc'$; the last inference of this must be NEW-RECSPLIT with the premises

$$sc' \text{ is a nonempty set,}$$

$$\text{for all } np \in sc' \text{ we have } D \vdash np \leq \&nrcs,$$

and

$$\begin{array}{l}\text{for all } c \text{ and } nk \text{ such that } \&nrcs \succeq c(np) \in \overline{D} \\ \text{there is an } s' \text{ such that} \\ \quad D \vdash nk \asymp s', \text{ and} \\ \quad \text{for all } np \text{ in } s' \text{ there is an } \&nkcs \in sc' \text{ and a } nq \text{ such that} \\ \quad\quad \&nkcs \succeq c(nq) \in \overline{D}, \text{ and} \\ \quad\quad D \vdash np \leq nq.\end{array}$$

Applying the last of these to nk and c gives an s' such that

$$D \vdash nk \asymp s'$$

and

$$\begin{array}{l}\text{for all } np \text{ in } s' \text{ there is an } \&nkcs \in sc' \text{ and a } nq \text{ such that} \\ \quad \&nkcs \succeq c(nq) \in \overline{D}, \text{ and} \\ \quad D \vdash np \leq nq.\end{array} \tag{3.28}$$

By Fact 3.36 (Recursive Split Intersection) on page 210,

$$D \vdash nk \ \& \ \text{rtort}(r) \asymp \{np \ \& \ \text{rtort}(r) \mid np \in s'\}.$$

Let $s = \{p \& r \mid \text{rtort}(p) \in s'\}$. By Fact 3.44 (Refinement and Recursive Split Consistency) on page 212, if we choose p such that $nk = \text{rtort}(p)$, we have $p \ \& \ r \asymp s$. Since $D \vdash \text{rtort}(r) \leq nk$, we have $p \ \& \ r \equiv r$, so EQUIV-SPLIT-L gives $r \asymp s$.

Let $k \in s$ be given; thus $k \propto p \& r$ where $\text{rtort}(p) \in s'$. By (3.28), there is an $\&nkcs \in sc'$ and a nq such that $\&nkcs \succeq c(nq) \in \overline{D}$ and $D \vdash \text{rtort}(p) \leq nq$. By Lemma 3.29 (Recursive Intersection Lower Bound) on page 196, this implies $D \vdash \text{rtort}(p) \ \& \ \text{rtort}(r) \leq nq$, and

the definition of rtort immediately gives $D \vdash \text{rtort}(k) \leq nq$. The definition of weakened intersection then gives

$$\&nkcs > c(\text{rtort}(k)) \in \widehat{D}$$

and the definition of $\overset{\text{def}}{:}$ gives $c \overset{\text{def}}{:} k \hookrightarrow \&nkcs$. Since $\&nkcs \in sc'$, $\wedge nkcs$ is in sc, so this is our conclusion. □

Assumption 2.51 (Constructor And Introduction) on page 67: If $c \overset{\text{def}}{:} r \hookrightarrow rc$ and $c \overset{\text{def}}{:} r \hookrightarrow kc$ then $c \overset{\text{def}}{:} r \hookrightarrow (rc \overset{\text{def}}{\wedge} kc)$.

Proof: The proof is very straightforward despite its length and may be skipped on the first reading.

If c is old, then Constructor And Introduction continues to be true for it as it was before we added the declaration D.

If we infer either of our hypotheses because $D \vdash \text{rtort}(r)$ empty, then the definition of $\overset{\text{def}}{:}$ gives our conclusion immediately.

Otherwise, $rc \propto \&nrcs$ and $kc \propto \&nkcs$ and by the definition of $\overset{\text{def}}{:}$ we have $\&nrcs > c(\text{rtort}(r)) \in \widehat{D}$ and $\&nkcs > c(\text{rtort}(r)) \in \widehat{D}$. By definition of weakened intersection, there are np_1, $npcs_1$, np_2, and $npcs_2$ such that all of the following are true:

$$\begin{gathered}
\&npcs_1 \succeq c(np_1) \in \overline{D} \\
D \vdash \text{rtort}(r) \leq np_1 \\
D \vdash \&npcs_1 \leq \&nrcs \\
\&npcs_2 \succeq c(np_2) \in \overline{D} \\
D \vdash \text{rtort}(r) \leq np_2 \\
D \vdash \&npcs_2 \leq \&nkcs.
\end{gathered}$$

The definition of intersection membership gives

$$\&(npcs_1 \cup npcs_2) \succeq c(np_1 \,\&\, np_2) \in \overline{D}.$$

Fact 3.30 (Recursive Intersection Greatest) on page 198 gives

$$D \vdash \text{rtort}(r) \leq np_1 \,\&\, np_2$$

and Lemma 3.29 (Recursive Intersection Lower Bound) on page 196 gives

$$D \vdash \&(npcs_1 \cup npcs_2) \leq \&nrcs$$

and

$$D \vdash \&(npcs_1 \cup npcs_2) \leq \&nkcs$$

and Fact 3.30 (Recursive Intersection Greatest) on page 198 then gives

$$D \vdash \&(npcs_1 \cup npcs_2) \leq \&(nrcs \cup nkcs).$$

Finally the definition of weakened intersection then gives

$$\&(nrcs \cup nkcs) > c(\text{rtort}(r))$$

and the definitions of $\overset{\text{def}}{:}$ and $\overset{\text{def}}{\wedge}$ give $c \overset{\text{def}}{:} r \hookrightarrow rc \overset{\text{def}}{\wedge} kc$, which is our conclusion. □

Assumption 2.52 (Constructor Argument Strengthen) on page 67: If $c \overset{\text{def}}{:} r \hookrightarrow rc$ and $k \leq r$ then $c \overset{\text{def}}{:} k \hookrightarrow rc$.

Proof: If we inferred $c \overset{\text{def}}{:} r \hookrightarrow rc$ because $D \vdash \text{rtort}(r)$ empty, then Theorem 3.32 (Empty Transitivity) on page 200 gives $D \vdash \text{rtort}(k)$ empty, and our conclusion follows immediately. Otherwise our conclusion follows from Theorem 3.33 (Subtype Transitivity) on page 201. □

Assumption 2.53 (Constructor Result Weaken) on page 67: If $c \overset{\text{def}}{:} r \hookrightarrow rc$ and $rc \overset{\text{def}}{\leq} kc$, then $c \overset{\text{def}}{:} r \hookrightarrow kc$.

Proof: If we can infer $c \overset{\text{def}}{:} r \hookrightarrow rc$ because $D \vdash \text{rtort}(r)$ empty, then we get our conclusion immediately. Otherwise it follows from Theorem 3.33 (Subtype Transitivity) on page 201. □

Assumption 3.12 (Emptyness Constructor) on page 184: If $rc \overset{\text{def}}{\text{empty}}$ and $c \overset{\text{def}}{:} r \hookrightarrow rc$ then $\vdash r$ empty.

Proof: By definition of $c \overset{\text{def}}{:} r \hookrightarrow rc$, either $D \vdash \text{rtort}(r)$ empty or $\text{rtort}(rc) > c(\text{rtort}(r)) \in \widehat{D}$. If the former is true, then Fact 3.52 (Emptyness Consistency) on page 215 gives our conclusion. Otherwise the statement of NEW-INFER-EMPTY, two uses of Theorem 3.32 (Empty Transitivity) on page 200, and Fact 3.52 (Emptyness Consistency) on page 215 give our conclusion. □

Assumption 3.13 (Emptyness Subtyping) on page 185: If $rc \overset{\text{def}}{\text{empty}}$ and $kc \overset{\text{def}}{\leq} rc$ then $kc \overset{\text{def}}{\text{empty}}$. Immediate from Theorem 3.32 (Empty Transitivity) on page 200.

3.6.3 Value Containment

Here we will show that if a value has a refinement type, it has the corresponding recursive type.

Theorem 3.53 (Value Containment) *Under the assumptions introduced by D, if $\cdot \vdash v : r$ then $D \vdash v \in \text{rtort}(r)$.*

Proof: Take D as given, and let F be the natural encoding of the rules for recursive type membership in Figure 3.2 as a function, and let $Q = \{(v, \text{rtort}(r)) \mid \cdot \vdash v : r\}$. We need to show $Q \subset \text{gfp}(F)$ and by co-induction it suffices to show $Q \subset F(Q)$. Proof is by cases on

a pair (v, r) in Q. The proof is straightforward and is included only because the result is important.

Case: $r \propto r_1 \,\&\, r_2$ Then rtort(r) = rtort(r_1) & rtort(r_2). We can use WEAKEN-TYPE and $\cdot \vdash v : r$ to infer $\cdot \vdash v : r_1$ and $\cdot \vdash v : r_2$. Thus, by definition of Q, we have

$$(v, \text{rtort}(r_1)) \in Q$$

and

$$(v, \text{rtort}(r_2)) \in Q.$$

Then AND-RECVALUE gives $(v, \text{rtort}(r_1) \,\&\, \text{rtort}(r_2)) \in F(Q)$, which is what we wanted to show.

Case: $v \propto c\ \ v'$, c is new, and $r \propto nrc$ By Lemma 2.68 (Subtype Irrelevancy) on page 88 and $\cdot \vdash v : nrc$ we have

$$\cdot \Vdash v : nrc.$$

The last inference of this must be CONSTR-TYPE with the premises

$$c \stackrel{\text{def}}{:} k \hookrightarrow nrc \tag{3.29}$$

and

$$\cdot \vdash v' : k.$$

If we were able to infer (3.29) because $D \vdash$ rtort(k) empty, Fact 3.52 (Emptyness Consistency) on page 215 gives $\vdash k$ empty, and Fact 3.14 (Soundness of Refinement Type Empty) on page 185 contradicts $\cdot \vdash v' : k$. Thus we must have inferred (3.29) from $nrc > c(\text{rtort}(k))$. By Lemma 3.47 (Weakened Intersection Simplification I) on page 213, either $D \vdash$ rtort(k) empty or there is a np such that

$$D \vdash \text{rtort}(k) \leq np$$

and

$$nrc \succeq c(np) \in D.$$

We have already show that $D \vdash$ rtort(k) empty cannot be true. Thus the other branch of the disjunction is true, so we can choose a p such that rtort$(p) = np$. By Fact 3.42 (Refinement and Recursive Subtyping Equivalence) on page 212 we have $k \leq p$, and then WEAKEN-TYPE gives $\cdot \vdash v' : p$. Thus by definition of Q we have $(v', \text{rtort}(p)) \in Q$, and NEW-RC-RECVALUE then gives $(c\ \ v', nrc) \in F(Q)$, which is our conclusion.

Case: $v \propto c\ \ v'$, c is new, $r \propto \&nrcs$ where $nrcs$ has two or more elements.

WEAKEN-TYPE immediately gives

$$\text{for all } nrc \in nrcs \text{ we have } \cdot \vdash v : \&\{nrc\}.$$

Then the definition of Q gives

$$\text{for all } nrc \in nrcs \text{ we have } (v, nrc) \in Q$$

and AND-RECVALUE then gives $(v, \&nrcs) \in F(Q)$, which is our conclusion.

Case: $v \propto c\ \ v'$, c is old, $r \propto rc$ Then we can immediately use OLD-RC-RECVALUE to get $(v, rc) \in F(Q)$, which is our conclusion.

Case: $v \propto \texttt{fn}\ \ x : t\ \texttt{=>}\ \ e$ and $r \propto r_1 \to r_2$ Suppose $\cdot \vdash v'' : r_1$ and $(\texttt{fn}\ \ x : t\ \texttt{=>}\ \ e)\ \ v'' \Rightarrow v'$. Then, by APPL-TYPE, we have $\cdot \vdash (\texttt{fn}\ \ x : t\ \texttt{=>}\ \ e)\ \ v'' : r_2$, and by Theorem 2.71 (Refinement Type Soundness) on page 99 we have $\cdot \vdash v' : r_2$. The definition of Q then gives $(v', \text{rtort}(r_2)) \in Q$.

Thus, by ABS-RECVALUE we have $((\texttt{fn}\ \ x : t\ \texttt{=>}\ \ e), nr \to \text{rtort}(r_2)) \in F(Q)$. By definition of rtort, this is our conclusion.

Case: $v \propto (v_1, \ \ldots, \ v_n)$ and $r \propto r_1 * \ldots * r_n$ By Lemma 2.68 (Subtype Irrelevancy) on page 88 we have $\cdot \Vdash (v_1, \ \ldots, \ v_n) : r_1 * \ldots * r_n$. The last inference of this must be TUPLE-TYPE with the premises

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } \cdot \vdash v_i : r_i.$$

By definition of Q this implies

$$\text{for } i \text{ in } 1 \ldots n \text{ we have } (v_i, \text{rtort}(r_i)) \in Q,$$

and the definition of rtort and TUPLE-RECSUB give $((v_1, \ \ldots, \ v_n), \text{rtort}(r)) \in F(Q)$, which is our conclusion. □

Chapter 4

Refinement Type Variables

There are two changes that need to be made to add polymorphism to refinement types. We need to add type variables and refinement type constructors that take type arguments. This chapter discusses the former, and Chapter 5 discusses the latter.

Adding the type variables is fairly simple. After looking at the various plausible options in Section 4.1, we will conclude that it seems best for each ML type variable to have exactly one refinement, which is a refinement type variable. Then in Section 4.2 we will describe the changes we must make to the machinery in Chapters 2 and 3 to accommodate this.

4.1 Adding Type Variables

The motivating examples behind refinement types have not made interesting use of polymorphism so far. Since ML is indeed a polymorphic language, we need a straightforward correct way to deal with polymorphism. For some examples, the desired behavior is clear. For instance, since `true` has the refinement type tt and `false` has the refinement type ff, we would like the statement

```
let val id = fn x => x
in
   (id true, id false)
end
```

to get the refinement type $tt * ff$.

Notation to represent the type for `id` seems straightforward. The most general ML type scheme for `id` is written as

$$\forall(\alpha).\alpha \rightarrow \alpha$$

By analogy with the notation for the ML type scheme, we shall also write the refinement type scheme as

$$\forall(\alpha).\alpha \rightarrow \alpha.$$

4.1.1 Instantiation

Next we need to understand how to instantiate the refinement type. Instantiating the ML type is simple; ML type inference tells us that in this example the α in the ML type scheme should be instantiated to $bool$, giving a resulting type of $bool \to bool$. Instantiating the refinement type scheme is a little more complex because it must have both of the types $tt \to tt$ and $ff \to ff$. We could take the path used in [Pie91a] and instantiate the refinement type scheme to a list of refinement types that is explicitly given in the expression; in this case we would be instantiating the α in $\forall(\alpha).\alpha \to \alpha$ to $\{tt, ff\}$ to get the type $tt \to tt \wedge ff \to ff$. However, we know that all of the types in the list must be refinements of $bool$, and since $bool$ has finitely many refinements, we can put all of those refinements into the list. We might as well put all possible refinements into the list since that will yield the most precise possible result; since there is therefore only one reasonable list, we can omit the list instead of explicitly specifying it.

Thus, to instantiate a refinement type scheme, we instantiate each refinement type variable to an ML type. The result is the intersection of all distinct results of substituting refinements of that ML type for the refinement type variable. For example, instantiating the refinement type variable α to $bool$ in the refinement type scheme $\forall(\alpha).\alpha \to \alpha$ yields

$$tt \to tt \wedge ff \to ff \wedge \top_{bool} \to \top_{bool} \wedge \bot_{bool} \to \bot_{bool}$$

Once we move on to more complex examples, more choices arise. Consider the expression skeleton:

```
let val strictif = fn x => fn y => fn z => if x then y else z
in
   ...
end
```

What should be the refinement type scheme for `strictif`? The ML scheme is

$$\forall(\alpha).bool \to \alpha \to \alpha \to \alpha.$$

It is tempting to permit multiple refinements of each ML type variable, since we could give this expression an informative type scheme like

$$\begin{array}{rl} \forall(\alpha,\beta).tt \to \alpha \to \beta \to \alpha & \wedge \\ ff \to \alpha \to \beta \to \beta & \wedge \\ \top_{bool} \to \alpha \to \alpha \to \alpha & \wedge \\ \bot_{bool} \to \alpha \to \alpha \to \bot_{\alpha} & \end{array}$$

Unfortunately, if we permit arbitrarily many refinements of each ML type variable, then there are expressions with no principal type. We will illustrate this with an example.

In the example, we shall use the datatype

$$\texttt{datatype}\ \alpha\ \mathit{option}\ \texttt{= SOME of}\ \alpha\ \texttt{| NONE}$$

to represent a value which may or may not be present. Note that `NONE` has no argument; we are using the concise version of the syntax for this example. We want to distinguish the two constructors of this datatype at the refinement type level, so we use the following rectype statement:

$$\begin{array}{l}\texttt{rectype}\ \alpha\ \mathit{some}\ \texttt{= SOME}\ (\alpha) \\ \quad\texttt{and}\ \alpha\ \mathit{none}\ \texttt{= NONE}\end{array}$$

(Although the value constructor `NONE` takes no argument, the refinement type constructor $none$ does take one argument because it refines the ML type constructor $option$ which takes one argument.)

Suppose we have a collection of rewrite rules which only apply sometimes. We can represent one of these rules as a function with the type

$$\alpha \to \alpha\ option$$

If the rewrite rule `rew` applies to a value x, then `rew` $x \Rightarrow$ `SOME` y, where y is the result of rewriting x. If the rewrite rule does not apply, then `rew` $x \Rightarrow$ `NONE`.

One natural thing to do with a rewrite rule to repeat it until it no longer applies; the result is a rewrite rule. We can write this as a straightforward higher-order function:

```
fun repeat rew x =
    case rew x of
       NONE => SOME x
     | SOME y => repeat rew y
```

which has the ML type $(\alpha \to \alpha\ option) \to \alpha \to \alpha\ option$.

Now, assuming an infinite number of refinement type variables refine each ML type variable, what is the type for `repeat`? Suppose the refinement type of the value bound to `x` is α_1. Perhaps the value bound to `x` will be rewritten zero times; this happens if the argument to `repeat` has type $\alpha_1 \to \alpha_2\ none$, so `repeat` has the type

$$(\alpha_1 \to \alpha_2\ none) \to \alpha_1 \to \alpha_1\ some$$

The value bound to `x` may also be rewritten once. This happens if the rewriter has type

$$\alpha_1 \to \alpha_2\ some \wedge \alpha_2 \to \alpha_3\ none,$$

so `repeat` also has the type

$$(\alpha_1 \to \alpha_2\ some \wedge \alpha_2 \to \alpha_3\ none) \to \alpha_1 \to \alpha_2\ some.$$

We can continue in this fashion, giving types for `repeat` that describe its behavior when the rewrite rule applies any nonnegative number of times. The only type that includes all of this information is the infinite intersection of all of these possible types. Our system only has finite refinement types, so `repeat` does not have a principal type if we permit infinitely many refinements of an ML type variable. The only natural choices for the number of refinements of each ML type variable seem to be one or infinitely many. Since we have ruled out having infinitely many, we shall have only one.

4.1.2 ML Polymorphism vs. Refinement Typing

Once we decide that each ML type variable has only one refinement (or any other fixed number), ML polymorphism interferes with refinement typing. For example, consider the expression

```
let val not = fn x => if x then false else true
    val double = fn f => fn x => f (f x)
in
   double not true
end
```

There are two ways to derive an ML type for this expression. Either we can have a polymorphic `double` with the type scheme

$$\forall(\alpha).(\alpha \to \alpha) \to \alpha \to \alpha$$

and instantiate α to $bool$ when `double` is used, or we can have a monomorphic `double` with the type scheme

$$\forall().(bool \to bool) \to bool \to bool$$

which does not need to be instantiated before it is used.

These two ways to derive an ML type for the above expression have different consequences for refinement typing. If `double` is polymorphic, then clearly the only refinement type scheme it can have is

$$\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

Instantiating α to the refinements of $bool$ gives an intersection with four components:

$$\begin{array}{ll}
(tt \to tt) \to tt \to tt & \wedge \\
(ff \to ff) \to ff \to ff & \wedge \\
(\top_{bool} \to \top_{bool}) \to \top_{bool} \to \top_{bool} & \wedge \\
(\bot_{bool} \to \bot_{bool}) \to \bot_{bool} \to \bot_{bool} &
\end{array}$$

If `double` has this refinement type, then what is the refinement type of `double not true`? The first two components do not help us determine this type, since

`not` has neither of the types $tt \to tt$ nor $ff \to ff$. The last component is irrelevant because `true` does not have the type $\bot_{bool}$. Thus the only usable component is the third and the best type for `double not true` is $\top_{bool}$.

If, on the other hand, `double` is monomorphic, then its type is much more informative. In the definition of `double`

```
fn f => fn x => f (f x)
```

we can assume that `f` has the type $tt \to ff \wedge ff \to tt$ and that `x` has the type tt. Then `f x` has the type ff, and `f (f x)` has the type tt, so the best type for `double not true` is tt.

Thus, the programmer using refinement types will have to have in mind the same derivation of the ML type that the compiler has in mind if he wants to predict what refinement types the compiler will assign to a particular expression. This should not be too difficult, since the algorithms actually used in the compilers always generate the most general ML type at every opportunity. Thus in the example above the programmer can expect the compiler to use the most general type for `double`, which results in the less informative refinement type for the `let` statement. If the programmer wanted the `let` statement to have the more informative type, he could use explicit ML type declarations to reduce the polymorphism. One way to do this would be to declare the second argument of `double` to have type $bool$, as in:

```
let val not = fn x => if x then false else true
    val double = fn f => fn x : bool => f (f x)
in
   double not true
end
```

In this thesis we avoid a choice among the various algorithms that could conceivably be used for ML type inference by stipulating that our terms must have enough embedded ML type information to uniquely determine how the ML type is derived. This requires us to at least explicitly specify which variables each `let` statement generalizes over. This isn't quite sufficient though; consider the let statement

```
let id = Λ(α). fn x => x
in
   id true
end
```

This expression has type $bool$. We could have derived this type by giving `id` the type scheme $\forall(\alpha).\alpha \to \alpha$ and instantiating α to $bool$ immediately before using `id`, or we could have derived it by giving `id` the unusual type scheme $\forall(\alpha).bool \to bool$ and instantiating α to something arbitrary before using `id`. To eliminate this ambiguity, we must have an explicit declaration each place a type variable could be introduced, which means an explicit

declaration each time a variable is used and for each abstraction. Because SML eagerly evaluates values bound by both `let` and `fn`, we can make our language more uniform by declaring an instantiation for every variable reference, including nonpolymorphic variables such as `x` in this example. Thus the fully explicit form of the above `let` statement is:

```
let id = Λ(α). fn x:α => x[]
in
   id[bool] true
end
```

The constructor `true` does not need the square brackets because it is not a variable and because constructors will not be polymorphic until Chapter 5. Constructors can be distinguished from variables because they are explicitly mentioned in the grammar for the language and in the inference rules.

4.2 Formally Incorporating Type Variables

We will write type variables as α, β; the mathematical variable that can stand for any type variable is written α; context should make it clear whether we are talking about one particular type variable or an arbitrary type variable.

The new grammars for ML types and refinement types have no surprises; we simply add productions for type variables:

$$t ::= tc \mid t * \ldots * t \mid tunit \mid t \rightarrow t \mid \alpha$$
$$r ::= r \wedge r \mid r \rightarrow r \mid rc \mid r * \ldots * r \mid runit \mid \alpha$$

We write a possibly empty vector of type variables as $\overline{\alpha}$. We also have vectors of refinement types $\overline{r}$, and so forth. We can substitute a vector of refinement types $\overline{r}$ for a vector of type variables $\overline{\alpha}$ in a refinement type r by using the notation $[\overline{r}/\overline{\alpha}]r$. The length of the vector $\overline{\alpha}$ is written length$(\overline{\alpha})$, and the i'th element is written $\overline{\alpha}\{i\}$. The first element is $\overline{\alpha}\{1\}$, not $\overline{\alpha}\{0\}$.

We define *ML type schemes* to have the form $\forall(\overline{\alpha}).t$, and *refinement type scheme*to have the form $\forall(\overline{\alpha}).r$.

The only changes we must make the object language grammar on page 19 is adding `let` statements and adding instantiations after each variable reference as discussed on page 228; the entire grammar is:

$$\begin{aligned} e ::= {} & x[\overline{t}] \mid \texttt{fn}\ x{:}t\ \texttt{=>}\ e \mid e\ \ e \mid c\ \ e \mid \\ & \texttt{case}\ e\ \texttt{of}\ c\ \texttt{=>}\ e\ \texttt{|}\ \ldots\ \texttt{|}\ c\ \texttt{=>}\ e\ \texttt{end}{:}t \mid \\ & (e, \ldots, e) \mid () \mid \texttt{elt\_}m\texttt{\_}n\ e \mid \\ & \texttt{fix}\ f{:}t\ \texttt{=>}\ \texttt{fn}\ x{:}t\ \texttt{=>}\ e \mid \\ & \texttt{let}\ x\ \texttt{=}\ \Lambda(\overline{\alpha}).e\ \texttt{in}\ e\ \texttt{end} \end{aligned}$$

```
let double = Λ(α).fn f:α→α => fn x:α => f[] (f[] x[])
in
   let not = Λ(). fn arg:bool =>
      case arg[] of
         True => fn _ => False ()
       | False => fn _ => True ()
      end:bool
   in
      double[bool] not[] (True ())
   end
end
```

Figure 4.1: Sample Expression Using Polymorphism

We define an *expression scheme* to have the form $\Lambda(\overline{\alpha}).e$, as appears in the grammar for `let` statements. If a variable is bound by a `let`, then the substitution after it specifies how to instantiate the expression scheme before use. If the variable is bound by a `fn`, then the substitution must be trivial (that is, both the vector of ML types and the vector of type variables must have zero length). For this chapter, we assume that our value constructors are still monomorphic. An example of the syntax is in Figure 4.1. The problem of converting human readable code into this syntax is simply ML type inference.

In Chapter 2, we defined substitution of closed expressions for variables in expressions. Because the expressions were closed and there were no type variables in the language at the time, the problem of variable capture did not arise. In the present case, we still limit substitution to expressions with no free object language variables, but they may have free type variables that can be bound by `let` statements; thus we now have to deal with the possibility of type variable capture during substitution. For example, the terms

```
let y = Λ(α).x[] in
   y[bool] (true ())
end
```
(4.1)

and

```
let y = Λ(β).x[] in
   y[bool] (true ())
end
```
(4.2)

have the same meaning in some intuitive sense. If we ignore the issue of variable capture while substituting `Λ().fn z:α => z []` for `x` in each of these, we get

```
let y = Λ(α).fn z:α => z [] in
   y[bool] (true ())
end
```

and

$$\begin{array}{l}\texttt{let y = } \Lambda(\beta).\texttt{fn z:}\alpha \texttt{ => z [] in} \\ \quad \texttt{y}[\mathit{bool}]\ (\texttt{true ()}) \\ \texttt{end.}\end{array} \tag{4.3}$$

Something has gone wrong here because these expressions no longer have intuitively equivalent meanings; in fact, in the ML type system we define below, the first has well-formed ML types but the second does not. The problem occurred when we substituted an expression with a free α into a context in which α was bound. There are several ways we could have prevented the problem.

First, we could simply forbid substitutions that cause variable capture. This would mean that (4.1) and (4.2) are no longer equivalent, because the substitution above is forbidden for (4.1) but permitted for (4.2). This is aesthetically unpleasing, but similar to approaches taken by others in the past; for example, with slightly different notations, the papers [Car87, CDDK86, DM82, Myc84, Tof88] all define systems that allow one to derive

$$f : \alpha \vdash \texttt{fn x => x} :: \forall\beta.\beta \to \beta$$

but not

$$f : \alpha \vdash \texttt{fn x => x} :: \forall\alpha.\alpha \to \alpha.$$

There are other approaches, such as higher-order abstract syntax[PE88, MNPS91, HHP93] and de Bruijnindices [Bar80, dB72], that solve the problem by changing or eliminating the notion of "named variable". These seem too radical for the task at hand.

Instead, we will circumvent the problem by giving a different meaning to (4.1) and (4.2) so they are actually the same mathematical object, as was done in [Bar80, page 26] and [CR36]. In this approach, we identify two expressions if we can transform one into the other by renaming bound variables, and whenever we write an expression, we really mean the equivalence class containing that expression. In this case the proper definition of substitution still forbids variable capture as a special case, but we can always find an element of the equivalence class that makes the substitution go through. With this interpretation, the correct result of the substitution mentioned above is (4.3). We use the same strategy to deal with binding object language variables; thus $\texttt{fn y:}\mathit{bool}\texttt{ => y}[]$ and $\texttt{fn x:}\mathit{bool}\texttt{ => x}[]$ are the same term, as are

$$\texttt{let x = } \Lambda(\alpha).\texttt{fn z:}\alpha \texttt{ => z}[]\ \texttt{in x}[\mathit{bool}]\ \texttt{end}$$

and

$$\texttt{let y = } \Lambda(\alpha).\texttt{fn z:}\alpha \texttt{ => z}[]\ \texttt{in y}[\mathit{bool}]\ \texttt{end.}$$

Now that we have a clear policy for dealing with type variable capture, we can modify the definition of substitution on page 23 so we now substitute expression schemes (not expressions) for variables in expressions. Only a few of the clauses of the substitution

definition change in a non-trivial way; the first clause listed uses the operation of substituting ML types for type variables in expressions, which is a simple operation we shall not formally define here:

$$
\begin{array}{l}
[\Lambda(\overline{\alpha}).e/x](x[\overline{t}]) = [\overline{t}/\overline{\alpha}]e \text{ if length}(\overline{\alpha}) = \text{length}(\overline{t}) \\
[\Lambda(\overline{\alpha}).e/x](x[\overline{t}]) \text{ fails otherwise} \\
[\Lambda(\overline{\alpha}).e/x](y[\overline{t}]) = y[\overline{t}] \text{ if } y \neq x \\
[\Lambda(\overline{\alpha}).e_1/x](\texttt{let } y \texttt{ = } \Lambda(\overline{\beta}).e_2 \texttt{ in } e_3 \texttt{ end}) = \\
\qquad (\texttt{let } y \texttt{ = } \Lambda(\overline{\beta}).[\Lambda(\overline{\alpha}).e_1/x]e_2 \texttt{ in } [\Lambda(\overline{\alpha}).e_1/x]e_3 \texttt{ end}) \\
\qquad \text{where } x \neq y \text{ and } \overline{\alpha} \text{ and } \overline{\beta} \text{ have no elements in common.}
\end{array}
$$

For example, substituting $\Lambda(\alpha).\texttt{fn x:}\alpha \texttt{ => x[]}$ for y in $\texttt{y}[bool]$ $(\texttt{true ()})$ yields

$$(\texttt{fn x:}bool \texttt{ => x[]})\ (\texttt{true ()}),$$

and substituting $\Lambda().\texttt{true ()}$ for x in $(\texttt{x[], x[]})$ yields $(\texttt{true (), true ()})$.

The grammar for values is unchanged, but the meaning changes slightly because the grammar references expressions, and expressions have changed:

$$v ::= c \ \ v \mid (v, \ \ldots, \ v) \mid () \mid \texttt{fn } x\texttt{:}t \texttt{ => } e$$

The changes to the evaluation relation defined in Figure 2.1 on page 24 consist of adding a rule for `let` and modifying rules that do substitution to construct trivial expression schemes so we can use the modified definition of substitution above:

$$\text{LET-SEM:} \quad \frac{\begin{array}{c} e_1 \Rightarrow v_1 \\ [(\Lambda(\overline{\alpha}).v_1)/x]e_2 \Rightarrow v_2 \end{array}}{\texttt{let } x \texttt{ = } \Lambda(\overline{\alpha}).e_1 \texttt{ in } e_2 \texttt{ end} \Rightarrow v_2}$$

$$\text{APPL-SEM:} \quad \frac{\begin{array}{c} e_1 \Rightarrow \texttt{fn } x\texttt{:}t \texttt{ => } e_3 \\ e_2 \Rightarrow v_2 \\ [(\Lambda().v_2)/x]e_3 \Rightarrow v_3 \end{array}}{e_1 \ \ e_2 \Rightarrow v_3}$$

$$\text{FIX-SEM:} \quad \frac{}{\begin{array}{l} \texttt{fix } f\texttt{:}t \texttt{ => fn } x\texttt{:}u \texttt{ => } e \Rightarrow \\ \qquad [(\Lambda().\texttt{fix } f\texttt{:}t \texttt{ => fn } x\texttt{:}u \texttt{ => } e)/f]\texttt{fn } x\texttt{:}u \texttt{ => } e \end{array}}$$

Notice that the `let-sem` rule says to eagerly evaluate the variable in `let` statements. There have been proposals to evaluate them lazily under some circumstances; we could do that with the alternative rule

$$\text{LET-SEM'}: \quad \frac{[(\Lambda(\overline{\alpha}).e_1)/x]e_2 \Rightarrow v_2}{\texttt{let } x \texttt{ = } \Lambda(\overline{\alpha}).e_1 \texttt{ in } e_2 \texttt{ end} \Rightarrow v_2}$$

Since we do not have polymorphic type constructors yet, our value constructors will not have polymorphic outputs. Thus it would be peculiar for them to have polymorphic inputs; for example, consider this declaration:

$$\texttt{datatype}\ \mathit{foo}\ \texttt{=}\ \texttt{Bar of}\ \alpha$$

Standard ML disallows `datatype` declarations where the constructors have type variables free in the input type (α in this example) that are not free in the output type (*foo* in this example), so we shall forbid them also. Since we cannot have a polymorphic output type in this chapter, we will outlaw type variables in the input type altogether:

Assumption 4.1 (Free Type Variables in Constructors) *If*

$$c \stackrel{\text{def}}{::} t \rightarrow \mathit{tc}$$

then t *has no type variables.*

The modifications to the ML type system introduce few surprises. The environment VM now maps variable names to ML type schemes. For uniformity, it maps all variable names to ML type schemes, even the variables bound by `fn` and `fix`; as the ABS-VALID rule below says, all such variables are bound to vacuous ML type schemes that quantify over zero type variables. We add a rule for `let` statements, and make slight modifications to the rules for variables, abstractions, and fixed points to accommodate the new environment:

$$\text{LET-VALID:}\quad \frac{\begin{array}{c}\text{VM} \vdash e_1 :: t_1 \\ \text{for all } \alpha \text{ in } \overline{\alpha} \text{ we have } \alpha \text{ is not free in VM} \\ \text{VM}[x := \forall(\overline{\alpha}).t_1] \vdash e_2 :: t\end{array}}{\text{VM} \vdash \texttt{let}\ x\ \texttt{=}\ \Lambda(\overline{\alpha}).e_1\ \texttt{in}\ e_2\ \texttt{end} :: t}$$

$$\text{VAR-VALID:}\quad \frac{\begin{array}{c}\text{VM}(x) = \forall(\overline{\alpha}).t \\ \text{length}(\overline{\alpha}) = \text{length}(\overline{t})\end{array}}{\text{VM} \vdash x[\overline{t}] :: [\overline{t}/\overline{\alpha}]t}$$

$$\text{ABS-VALID:}\quad \frac{\text{VM}[x := \forall().t_1] \vdash e :: t_2}{\text{VM} \vdash (\texttt{fn}\ x\texttt{:}t_1\ \texttt{=>}\ e) :: t_1 \rightarrow t_2}$$

$$\text{FIX-VALID:}\quad \frac{\text{VM}[f := \forall().t_1 \rightarrow t_2] \vdash (\texttt{fn}\ x\texttt{:}t_1\ \texttt{=>}\ e) :: t_1 \rightarrow t_2}{\text{VM} \vdash (\texttt{fix}\ f\texttt{:}t_1 \rightarrow t_2\ \texttt{=>}\ \texttt{fn}\ x\texttt{:}t_1\ \texttt{=>}\ e) :: t_1 \rightarrow t_2}$$

Fact 2.3 (ML Type Soundness) on page 27 and Lemma 2.4 (Unique Inferred ML Types) on page 27 still hold for the modified language, as do Fact 2.5 (ML Free Variables Bound) on page 29 and Fact 2.6 (ML Value Substitution) on page 29.

We augment the refinement rules in Figure 2.3 on page 31 by asserting that each refinement type variable refines the corresponding ML type variable:

$$\text{VAR-REF:}\quad \frac{}{\alpha \sqsubset \alpha}$$

We say $\overline{r} \sqsubset \overline{t}$ if length($\overline{r}$) = length($\overline{t}$) and, for i between 1 and length($\overline{r}$), we have $\overline{r}\{i\} \sqsubset \overline{t}\{i\}$. A refinement type scheme refines an ML type scheme if they quantify over the same variables and, after stripping off the quantifiers, the underlying refinement type refines the underlying ML type.

Lemma 2.10 (Unique ML Types) on page 31 is still true; the added case to the proof is trivial, and we shall omit it. We augment the definition of rtom on page 32 so it also works on substitutions mapping type variables to refinement types; for example,

$$\begin{array}{l}\text{rtom}([tt/\alpha, ff/\beta])(\texttt{fn x:}\alpha * \beta \texttt{ => x[]}) = \\ [bool/\alpha, bool/\beta](\texttt{fn x:}\alpha * \beta \texttt{ => x[]}) \quad = \\ \texttt{fn x:}\mathit{bool} * \mathit{bool} \texttt{ => x[]}.\end{array}$$

Fact 2.12 (Tuple Refines) on page 32 is still true.

We need to make no change to the subtyping rules in Figure 2.4 on page 35, since the SELF-SUB rule ensures that refinement type variables are subtypes of themselves. Similarly, we do not need to change the rules for splitting in Figure 2.5 on page 48 because the SELF-SPLIT rule deals with refinement type variables.

Now that we have both substitution and subtyping, we have to show that they interact with each other in the natural way:

Fact 4.2 (Type Substitution Preserves Subtyping) *If $r \leq k$, then for any well-formed substitution s, we have $s(r) \leq s(k)$.*

The proof of this is by induction on the derivation of $r \leq k$.

We also need to prove that substitution and splitting interact in a natural way:

Fact 4.3 (Split Substitution) *If $r \asymp \{r_1, \ldots, r_n\}$, then for any r' and α we have*

$$[r'/\alpha]r \asymp \{[r'/\alpha]r_1, \ldots, [r'/\alpha]r_n\}.$$

The proof of this is a simple induction on the derivation of the hypothesis.

The changes to refinement type inference are entirely analogous to the changes to ML type inference. The variable environment VR now contains refinement schemes rather than simple refinement types. Starting with the rules in Figure 2.6 on page 60, we add a rule for `let`, change the rule for variables, and make minor changes to the rules for abstractions

and fixed points:

$$\text{LET-TYPE:}\quad \frac{\begin{array}{c}\text{VR} \vdash e_1 : r_1 \\ \text{for all } \alpha \in \overline{\alpha} \text{ we have } \alpha \text{ not free in VR} \\ \text{VR}[x := \forall(\overline{\alpha}).r_1] \vdash e_2 : r\end{array}}{\text{VR} \vdash \texttt{let } x \texttt{ = } \Lambda(\overline{\alpha}).e_1 \texttt{ in } e_2 \texttt{ end} : r}$$

$$\text{VAR-TYPE:}\quad \frac{\begin{array}{c}\text{VR}(x) = \forall(\overline{\alpha}).r \\ \overline{r} \sqsubset \overline{t} \\ r \sqsubset t\end{array}}{\text{VR} \vdash x[\overline{t}\,] : [\overline{r}/\overline{\alpha}]r}$$

$$\text{ABS-TYPE:}\quad \frac{\text{VR}[x := \forall().r] \vdash e : k \qquad r \sqsubset t}{\text{VR} \vdash \texttt{fn } x : t \texttt{ => } e : r \to k}$$

$$\text{FIX-TYPE:}\quad \frac{\begin{array}{c}r \sqsubset t_1 \to t_2 \\ \text{VR}[f := \forall().r] \vdash (\texttt{fn } x : t_1 \texttt{ => } e) : r\end{array}}{\text{VR} \vdash (\texttt{fix } f : t_1 \to t_2 \texttt{ => fn } x : t_1 \texttt{ => } e) : r}$$

In the syntax example in Figure 4.1 on page 229, using the LET-TYPE rule on the outer `let` statement leads us to add the type scheme

$$\forall(\alpha).(\alpha \to \alpha) \to \alpha \to \alpha$$

for `double` to the type environment before inferring a type for the inner `let` statement. Then we add the trivial type scheme

$$\forall().tt \to tt \wedge ff \to ff \wedge \bot_{bool} \to \bot_{bool} \wedge \top_{bool} \to \top_{bool}$$

for `not`. The only way we can instantiate `double` that allows the application

$$(\texttt{double}[bool]\ \texttt{not}[])\ (\texttt{True}\ ())$$

to have a type is to substitute $\top_{bool}$ for α; if we do this,

$$\texttt{double}[bool]$$

gets the type $(\top_{bool} \to \top_{bool}) \to \top_{bool} \to \top_{bool}$,

$$\texttt{double}[bool]\ \texttt{not}[]$$

gets the type $\top_{bool} \to \top_{bool}$, and

$$(\texttt{double}[bool]\ \texttt{not}[])\ (\texttt{True}\ ())$$

and the `let` statement as a whole has the type $\top_{bool}$.

Compatibility with ML is unaffected by the new rules added. We will omit the simple case for `let` that must be added to the proof of Theorem 2.54 (Inferred Types Refine) on page 68, and the proofs of Lemma 2.55 (Value Arrow Type) on page 74 and Fact 2.56 (Value Constructor Type) on page 74 are unchanged. We must add this equation to the definition of mtor on page 76:

$$\text{mtor}(\alpha) = \alpha;$$

the proofs of Fact 2.61 (mtor Refines) on page 76 and Fact 2.62 (Unique Refinement) on page 76 are still trivial, and the case we must add to deal with `let` statements in Theorem 2.64 (ML Compatibility) on page 77 is simple and we will omit it.

Updating the soundness proof is somewhat more work, and constitutes most of the remainder of this chapter. The statement of Lemma 2.66 (Environment Modification) on page 81 does not change; the VAR-TYPE case of the proof changes slightly: The statement of Lemma 2.66 (Environment Modification) on page 81 does not change; the proof only changes slightly:

Lemma 4.4 (Environment Modification) *If*

$$\text{VR} \vdash e : r$$

and

$$\text{VR}' \textit{ has the same domain as } \text{VR}$$

and

$$\textit{for } x \textit{ free in } e \textit{ we have } \text{VR}'(x) \leq \text{VR}(x)$$

then

$$\text{VR}' \vdash e : r.$$

Also, if in addition

$$\text{VR} \Vdash e : r$$

and

$$e \textit{ is not a variable}$$

then

$$\text{VR}' \Vdash e : r.$$

Proof: By induction on the derivation of $\text{VR} \vdash e : r$.

Case: VAR-TYPE Then e has the form $x[\overline{t}]$, and r has the form $[\overline{r}/\overline{\alpha}]k$ where the premises of VAR-TYPE are:

$$\begin{aligned} \text{VR}(x) &= \forall(\overline{\alpha}).k \\ \overline{r} &\sqsubset \overline{t} \\ k &\sqsubset t. \end{aligned} \tag{4.4}$$

Since $\text{VR}(x) \leq \text{VR}'(x)$, we know that

$$\text{VR}'(x) \propto \forall(\overline{\alpha}).k' \tag{4.5}$$

for some $k' \leq k$. By the version of Theorem 2.21 (Subtypes Refine) on page 36 that holds for this system, $k' \sqsubset t$. Then we can use VAR-TYPE with the premises (4.5), (4.4), and $k' \sqsubset t$ to get

$$\text{VR}' \vdash x[\overline{t}] : [\overline{r}/\overline{\alpha}]k'.$$

Then Fact 4.2 (Type Substitution Preserves Subtyping) on page 233 gives $[\overline{r}/\overline{\alpha}]k' \leq [\overline{r}/\overline{\alpha}]k$, and WEAKEN-TYPE then gives $\text{VR}' \vdash x[\overline{t}] : [\overline{r}/\overline{\alpha}]k$, which is our conclusion.

Case: Otherwise. Omitted. □

Lemma 2.67 (Piecewise Intersection) on page 84 and Lemma 2.68 (Subtype Irrelevancy) on page 88 are unchanged because values do not have `let` statements at the top level. Theorem 2.69 (Splitting Value Types) on page 89 is changed slightly because it has to use Fact 4.3 (Split Substitution) on page 233.

Lemma 2.70 (Value Substitution) on page 93 becomes slightly more interesting. We have to add a case for `let` declarations and make nontrivial changes to the case for variables. The new type inference rule for variables specifies a refinement type substitution, so we need to be able to substitute refinement types for type variables in refinement type derivations:

Fact 4.5 (Refinement Type Substitution) *Suppose* $\text{VR} \vdash e : r$ *and* s *is a substitution mapping type variables to refinement types where for all* α *in the domain of* s *we have* $s(\alpha)$ *is well formed. Then*

$$s(\text{VR}) \vdash \text{rtom}(s)(e) : s(r).$$

The proof of this is a simple induction on the derivation of $\text{VR} \vdash e : r$. The SPLIT-TYPE case uses Fact 4.3 (Split Substitution) on page 233.

This fact is useful in variable case of Value Substitution; we shall restate that lemma and give the `let` and variable cases of the proof:

Lemma 4.6 (Value Substitution) *If*

$$\text{VR} \vdash e_1 : r_1,$$

where e_1 *is a value or a closed expression of the form* `fix` $f : t_1$ `=>` `fn` $x : t_2$ `=>` e''*, and*

$$\text{VR}[x := \forall(\overline{\alpha}).r_1] \vdash e_2 : r_2,$$

and none of the variables in $\overline{\alpha}$ *are free in* VR*, and the substitution* $[(\Lambda(\overline{\alpha}).e_1)/x]e_2$ *succeeds, then*

$$\text{VR} \vdash [(\Lambda(\overline{\alpha}).e_1)/x]e_2 : r_2.$$

Proof: By induction on the derivation of $\text{VR}[x := \forall(\overline{\alpha}).r_1] \vdash e_2 : r_2$.

Case: VAR-TYPE Then e_2 has the form $y[\ldots]$ for some y. If y is not x, then the desired conclusion is $\text{VR} \vdash e_2 : r_2$. We can get this by eliminating the unused variable x from the hypothesis $\text{VR}[x := \forall(\overline{\alpha}).r_1] \vdash e_2 : r_2$.

Otherwise, e_2 has the form $x[\overline{t}]$. By the shape of VAR-TYPE, r_2 has the form $[\overline{r}/\overline{\alpha}]r_1$. The premises of VAR-TYPE must be $\overline{r} \sqsubset \overline{t}$ and, for some t, $r_1 \sqsubset t$. By Fact 4.5 (Refinement Type Substitution) on page 236,

$$[\overline{r}/\overline{\alpha}](\text{VR}) \vdash \text{rtom}([\overline{r}/\overline{\alpha}])(e_1) : [\overline{r}/\overline{\alpha}](r_1). \tag{4.6}$$

By hypothesis, none of the variables in $\overline{\alpha}$ are free in VR, so $[\overline{r}/\overline{\alpha}](\text{VR}) = \text{VR}$. By definition of substitution, $[(\Lambda(\overline{\alpha}).e_1)/x]e_2 = [(\Lambda(\overline{\alpha}).e_1)/x](x[\overline{t}]) = [\overline{t}/\overline{\alpha}]e_1 = \text{rtom}([\overline{r}/\overline{\alpha}])(e_1)$. Since $r_2 = [\overline{r}/\overline{\alpha}]r_1$, (4.6) is our conclusion.

Case: LET-TYPE Then e_2 has the form `let` y `=` $\Lambda(\overline{\beta}).e_3$ `in` e_4 `end` for some y. Rename variables if necessary to ensure that y and x are distinct, and that $\overline{\alpha}$ and $\overline{\beta}$ have no variables in common. For some r_3, the premises of `let-type` must be

$$\text{VR}[x := \forall(\overline{\alpha}).r_1] \vdash e_3 : r_3,$$

$$\text{for all } \beta \in \overline{\beta} \text{ we have } \beta \text{ not free in } \text{VR}[x := \forall(\overline{\alpha}).r_1],$$

and

$$\text{VR}[x := \forall(\overline{\alpha}).r_1, y := \forall(\overline{\beta}).r_3] \vdash e_4 : r_2.$$

Let L abbreviate $\Lambda(\overline{\alpha}).e_1$; then our induction hypothesis gives

$$\text{VR} \vdash [L/x]e_3 : r_3$$

and

$$\text{VR}[y := \forall(\overline{\beta}).r_3] \vdash [L/x]e_4 : r_2.$$

Then LET-TYPE gives

$$\text{VR} \vdash \texttt{let}\ y\ \texttt{=}\ \Lambda(\overline{\beta}).[L/x]e_3\ \texttt{in}\ [L/x]e_4 : r_2;$$

by definition of substitution, this is our conclusion.

Case: Otherwise Omitted. □

We are finally able to give the modifications of Theorem 2.71 (Refinement Type Soundness) on page 99 necessary for the system described in this chapter. Although the statement of the theorem does not change, we will repeat it here:

Theorem 4.7 (Refinement Type Soundness) *If* $e \Rightarrow v$ *and* $\cdot \vdash e : r$*, then* $\cdot \vdash v : r$.

Since e is closed, it is not a variable; thus the only change we need to make to the previous version of the proof is to deal with the possibility that e may be a `let` statement.

Proof: By cases on the pair (root inference of $e \Rightarrow v$, root inference of $\cdot \vdash e : r$).

Case: (LET-SEM, LET-TYPE) Then e must have the form `let` x `=` $\Lambda(\overline{\alpha}).e_1$ `in` e_2 `end`. The premises of LET-SEM must be

$$e_1 \Rightarrow v_1 \tag{4.7}$$

and

$$[(\Lambda(\overline{\alpha}).v_1)/x]e_2 \Rightarrow v. \tag{4.8}$$

For some r, the premises of LET-TYPE must include

$$\cdot \vdash e_1 : r_1 \tag{4.9}$$

and

$$\mathrm{VR}[x := \forall(\overline{\alpha}).r_1] \vdash e_2 : r. \tag{4.10}$$

Using the induction hypothesis on (4.7) and (4.9) gives

$$\cdot \vdash v_1 : r_1$$

Value Substitution and (4.10) then give

$$\cdot \vdash [(\Lambda(\overline{\alpha}).v_1)/x]e_2 : r.$$

Using the induction hypothesis on this and (4.8) gives $\cdot \vdash v : r$, which is our conclusion.

Case: Otherwise Omitted. □

The proofs of Theorem 2.90 (Finite Refinements) on page 115 and Corollary 2.91 (Principal Refinement Types) on page 115 are essentially unchanged.

The only significant change to the decision procedure is modifying the `infer` function defined in Figures 2.7 and 2.8 on pages 142 and 143 to deal with `let` statements and the new syntax for variable references. The new cases are:

```
fun infer VR y[t̄] =
      if VR(y) is undefined or
         VR(y) does not have the form ∀(ᾱ).r
            where r refines some t
      then ns
      else
         let val ∀(ᾱ).r = VR(y)
         in
            △fn {[r̄/ᾱ]r | length(r̄) = length(ᾱ) and
                        for all i we have r̄{i} ∈allrefs(t̄{i})}
         end
  | infer VR (e as (let x = Λ(ᾱ).e1 in e2 end)) =
      let val k = infer VR e1
          val s = split (k)
          val u = the unique u such that rtom(VR) ⊢ e :: u
          in
             sjoinf u {infer (VR[x := ∀(ᾱ).p]) e2 | p ∈ s}
          end
```

The correctness proof for the revised case for variables has no surprises, and the proof for the `let` case uses no concepts that do not appear in the application or abstraction cases, so we shall omit them.

By Assumption 4.1 (Free Type Variables in Constructors) on page 232, there cannot be any type variables in `rectype` declarations. Thus the argument in Chapter 3 needs no revisions to accommodate the type variables introduced in this chapter.

To summarize, once we decide that each ML type variable is refined by exactly one refinement type variable, the formal description of refinement types with type variables follows straightforwardly.

Chapter 5

Polymorphic Refinement Type Constructors

Programmers intuitively know that all even length lists of `true`'s are also even length lists of booleans. With polymorphic refinement type constructors, we can write this as $tt\ ev \leq \top_{bool}\ ev$, where ev is the type of lists with even length. This chapter is about formalizing that intuition in the type system.

We say that the type argument to ev is a *positive* type argument, since as the type argument of ev gets larger, the type as a whole gets larger. There are other possibilities; for example, suppose we have the declaration

$$\begin{array}{l}\texttt{datatype}\ \alpha\ \ pred\ \texttt{=}\ \texttt{Pred of}\ \alpha \to bool \\ \texttt{rectype}\ \alpha\ \ tpred\ \texttt{=}\ \texttt{Pred}\ (\alpha \to tt) \\ \quad\ \texttt{and}\ \alpha\ \ fpred\ \texttt{=}\ \texttt{Pred}\ (\alpha \to ff)\end{array}$$

Using an intuitive reading of this `rectype` declaration, we would expect to be able to apply `Pred` to a function with type $tt \to tt$ and get a value of type $tt\ tpred$; similarly, we would expect to be able to apply `Pred` to a function with type $\top_{bool} \to tt$ and get a value of type $\top_{bool}\ tpred$. Since $\top_{bool} \to tt \leq tt \to tt$, we expect $\top_{bool}\ tpred \leq tt\ tpred$. We say the type arguments of $pred$, $tpred$, and $fpred$ are all *negative*.

There are two other possibilities. We can have a type variable that appears on both the left and the right side of an arrow, such as

$$\texttt{datatype}\ \alpha\ \ m\ \texttt{=}\ \texttt{M of}\ \alpha \to \alpha\texttt{,}$$

where $tt\ \top_m$ is incomparable with $\top_{bool}\ \top_m$. We say the type argument of m is *mixed*.

We can also have type variables that appear nowhere in the type, such as

$$\texttt{datatype}\ \alpha\ \ i\ \texttt{=}\ \texttt{B of}\ bool$$

where $tt\ \top_i \equiv \top_{bool}\ \top_i$. We say the type argument of i is *ignored*. It turns out that positive, negative, mixed, and ignored are all the possibilities.

Since our type system is an approximation instead of an all-knowing oracle, we have the option of ignoring any or all of the above distinctions, so long as the resulting approximation is conservative. We could take the least informative approximation in all cases; this would mean treating all type arguments as mixed type arguments. In this case the refinement types $tt\ ev$ and $\top_{bool}\ ev$ would be incomparable. With this interpretation, $tt\ ev$ would no longer be a principal type of `cons (true, cons (true, nil))`, since the type $tt\ ev \wedge \top_{bool}\ ev$ would be another type for that expression that is strictly smaller. This approach has not been explored enough to determine how many unpleasant surprises it gives the programmer, but nevertheless we will not go that way.

Another approach is to simplify things by outlawing some of the possibilities; since all of the possibilities are permitted in Standard ML, this implies becoming less compatible with Standard ML. The current implementation does this; it outlaws mixed type arguments and it treats ignored type arguments as though they were positive. However, in this chapter we will permit and accurately model all four possibilities.

Thus, in general, each polymorphic refinement type constructor will have four kinds of type arguments. We will represent the different kinds by grouping them together, separated by semicolons, in the order negative, positive, mixed, and ignored. For example, the true form of $tt\ ev$ is $(;tt;;)\ ev$.

Each ML type constructor takes a fixed number of type arguments; each of these is either negative, positive, mixed, or ignored. We will assume these type arguments are grouped as described in the previous paragraph, so we can describe the number of arguments an ML type constructor takes with a tuple of four nonnegative integers saying how many arguments it takes of each type. We call this tuple the *arity* of the ML type constructor, and if we call the ML type constructor tc then we write its arity as $\text{arity}(tc)$. For instance, $\text{arity}(list) = (0;1;0;0)$ and $\text{arity}(m) = (0;0;1;0)$. Assumption 2.2 (Constructors have Unique ML Types) on page 26 still holds, so we can assume that the arity function is defined for ML type constructors, and define $\text{arity}(rc)$ to be $\text{arity}(tc)$ for the unique tc such that $rc \stackrel{\text{def}}{\sqsubset} tc$.

With these conventions, we can define arrow and tuple types as uses of ordinary type constructors. With this interpretation, many of the inference rules concerning tuple or arrow types are subsumed by more general rules. Specifically, we shall treat the "$\rightarrow$" operator that appears in ML types as an ordinary ML type constructor with arity $(1;1;0;0)$; we shall call it $tarrow$ when we are thinking of it in this context. We also have an "$\rightarrow$" operator in refinement types; we will call it $rarrow$, and we have the assumption $rarrow \stackrel{\text{def}}{\sqsubset} tarrow$. We will continue to use "$\rightarrow$", but now it is a readable abbreviation for a use of $rarrow$ or $tarrow$, rather than part of the syntax. For example $bool \rightarrow bool$ is syntactic sugar for $(bool;bool;;)\ tarrow$ and $tt \rightarrow ff$ is syntactic sugar for $(tt;ff;;)\ rarrow$.

Similarly, we can represent tupling of ML types as an ordinary ML type constructor.

For example, we give the ML type constructor that takes arguments α, β, and γ and constructs $\alpha * \beta * \gamma$ the name $ttuple_3$ which has arity $(0;3;0;0)$. In general we have an ML type constructor $ttuple_n$ for tuples of each nonnegative size n, and arity$(ttuple_n) = (0;n;0;0)$. Refinement type tuples get their own constructor, named $rtuple_n$, and for all nonnegative n we have $rtuple_n \stackrel{\text{def}}{\sqsubset} ttuple_n$. We will continue to use the old syntax as syntactic sugar for the new; for example, $runit$ stands for $(;;;)\ rtuple_0$ and $bool * bool$ stands for $(;bool,bool;;)\ ttuple_2$.

We will make the examples more readable by using some other syntactic sugar too. When a refinement or ML type constructor has no arguments, we eliminate the argument list entirely; thus we write $bool$ instead of $(;;;)\ bool$. Also, if all of the arguments are positive, we omit the semicolons, and if there is only one argument and that argument is positive, we omit the parentheses; thus we write $bool\ list$ instead of $(;bool;;)\ list$.

To make the rest of this chapter more concise, we will introduce special notation for groups of four vectors of types or type variables. We abbreviate $(\overline{r}_1;\overline{r}_2;\overline{r}_3;\overline{r}_4)$ as $\overline{\overline{r}}$. We define $\overline{\overline{t}}$ as a similar grouping of four $\overline{t}$'s.

With this said, it should not be surprising that after we expand all the syntactic sugar, the grammars for ML and refinement types have become simpler:

$$t ::= (\overline{\overline{t}})tc \mid \alpha$$
$$r ::= r \wedge r \mid (\overline{\overline{r}})rc \mid \alpha$$

We change the grammar for expressions by stating explicitly how to instantiate each value constructor before using it. We do this for the same reason we had an explicit instantiation after each variable in Chapter 4: we need the object language to uniquely determine the ML type derivation. We specify the ML types of value constructors with assumptions of the form

$$c \stackrel{\text{def}}{::} t \hookrightarrow (\overline{\overline{\alpha}})tc,$$

so the easiest way to specify the substitution is by giving a quadruple of types to substitute for the type variables $\overline{\overline{\alpha}}$. Thus the new grammar for expressions is:

$$\begin{aligned} e ::= {} & x[\overline{t}] \mid \texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ e \mid e\ \ e \mid c[\overline{\overline{t}}]\ \ e \mid \\ & \texttt{case}\ e\ \texttt{of}\ c\ \texttt{=>}\ e\ \texttt{|}\ \ldots\ \texttt{|}\ c\ \texttt{=>}\ e\ \texttt{end:}t \mid \\ & (e\texttt{,}\ \ldots\texttt{,}\ e) \mid () \mid \texttt{elt\_}m\texttt{\_}n\ \ e \mid \\ & \texttt{fix}\ f\texttt{:}t\ \texttt{=>}\ \texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ e \mid \\ & \texttt{let}\ x\ \texttt{=}\ \Lambda(\overline{\alpha}).e\ \texttt{in}\ e\ \texttt{end} \end{aligned}$$

As we did with type constructors, we may omit the substitution after a value constructor if it is empty; thus we will write `true` $()$ instead of `true`$[;;;]$ $()$.

Intersections of vectors of refinement types happen pointwise; that is, if $\overline{r}$ and $\overline{k}$ have the same length, then $\overline{r} \wedge \overline{k}$ has that length too, and $(\overline{r} \wedge \overline{k})\{i\} = \overline{r}\{i\} \wedge \overline{k}\{i\}$ for i between 1 and length$(\overline{r})$.

The arguments in this chapter are a modification to the arguments in Chapters 2 and 4. We will disregard trivial changes made to accommodate the change in syntax of the object language.

The definition of substitution on page 231 does not change, nor do the semantics rules on page 231.

5.1 ML typing

Now that we have polymorphic type constructors, we can have polymorphic `datatype` declarations. To permit this, we need to revise Assumption 4.1 (Free Type Variables in Constructors) on page 232:

Assumption 5.1 (Free Type Variables in Constructors) *If*

$$c \stackrel{\text{def}}{::} t \rightarrow (\overline{\overline{\alpha}})tc$$

then all type variables free in t *appear in* $\overline{\overline{\alpha}}$.

The only change to the appearance of the ML typing relation specified in Figure 2.2 on page 27 and updated on page 232 are to the rules for constructors and `case` statements:

CONSTR-VALID:
$$\frac{c \stackrel{\text{def}}{::} t \hookrightarrow (\overline{\overline{\alpha}})tc \qquad \text{VM} \vdash e :: [\overline{\overline{t}}/\overline{\overline{\alpha}}]t}{\text{VM} \vdash c[\overline{\overline{t}}]\ \ e :: (\overline{\overline{t}})tc}$$

CASE-VALID:
$$\frac{\begin{array}{c}\text{VM} \vdash e_0 :: (\overline{\overline{t}})tc \\ \text{for all } i \text{ we have } c_i \stackrel{\text{def}}{::} t_i \hookrightarrow (\overline{\overline{\alpha}})tc \\ \text{for all } i \text{ we have VM} \vdash e_i :: [\overline{\overline{t}}/\overline{\overline{\alpha}}]t_i \rightarrow u\end{array}}{\text{VM} \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}\, u) :: u}$$

The meaning of these rules have changed, though, because now $\rightarrow$ and $*$ are syntactic sugar for uses of polymorphic ML type constructors instead of primitive symbols. The modified system has all the usual properties; Fact 2.3 (ML Type Soundness) on page 27, Lemma 2.4 (Unique Inferred ML Types) on page 27, Fact 2.5 (ML Free Variables Bound) on page 29, and Fact 2.6 (ML Value Substitution) on page 29 still hold.

5.2 Subtyping

Because arrows and tuples are no longer primitive, we can eliminate some of rules for the refines relation "$\sqsubset$" defined in Figure 2.3 on page 31 and updated on page 233. The

$$\text{VAR-REF:}\quad \frac{}{\alpha \sqsubset \alpha}$$

$$\text{AND-REF:}\quad \frac{r_1 \sqsubset t \qquad r_2 \sqsubset t}{r_1 \wedge r_2 \sqsubset t}$$

$$\text{RCON-REF:}\quad \frac{rc \overset{\text{def}}{\sqsubset} tc \qquad \overline{\overline{r}} \sqsubset \overline{\overline{t}}}{[\overline{\overline{r}}]rc \sqsubset [\overline{\overline{t}}]tc}$$

$$\text{QUADRUPLE-REF:}\quad \frac{\overline{r}_1 \sqsubset \overline{t}_1 \qquad \overline{r}_2 \sqsubset \overline{t}_2 \qquad \overline{r}_3 \sqsubset \overline{t}_3 \qquad \overline{r}_4 \sqsubset \overline{t}_4}{(\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4) \sqsubset (\overline{t}_1; \overline{t}_2; \overline{t}_3; \overline{t}_4)}$$

$$\text{VECTOR-REF:}\quad \frac{\begin{array}{c}\text{length}(\overline{r}) = \text{length}(\overline{t}) \\ \text{for all } i \text{ in } 1 \ldots \text{length}(\overline{r}) \text{ we have } \overline{r}\{i\} \sqsubset \overline{t}\{i\}\end{array}}{\overline{r} \sqsubset \overline{t}}$$

Figure 5.1: Polymorphic Refinement Rules

complete set of rules is in Figure 5.1. In these rules we use the abbreviation $\overline{\overline{r}} \sqsubset \overline{\overline{t}}$ to mean that the quadruples $\overline{\overline{r}}$ and $\overline{\overline{t}}$ have the same shape and, for each r in $\overline{\overline{r}}$ and the corresponding t in $\overline{\overline{t}}$, we have $r \sqsubset t$. We are able to get the effect of the old ARROW-REF rule because we have the RCON-REF and we assume

$$rarrow \overset{\text{def}}{\sqsubset} tarrow.$$

Similarly, we get the effect of TUPLE-REF by using RCON-REF and the assumption

$$rtuple_n \overset{\text{def}}{\sqsubset} ttuple_n$$

for all nonnegative n.

Lemma 2.10 (Unique ML Types) on page 31 is still true, and we shall omit the proof. The definition of rtom on page 32 is unchanged, as is the addition that defines its effect on substitutions on page 233.

For the same reason, we can simplify the subtyping rules that originally appeared in Figure 2.4 on page 35. First we generalize RCON-SUB and RCON-AND-ELIM-SUB to deal with polymorphism, then we eliminate ARROW-SUB, ARROW-AND-ELIM-SUB, TUPLE-SUB, and TUPLE-AND-ELIM-SUB because those rules are now subsumed by the generalized RCON-SUB and RCON-AND-ELIM-SUB. The entire set of rules is in Figure 5.2.

To use RCON-SUB with arrows and tuples, we need arrow and tuple refinement type constructors to be subtypes of themselves; thus we need to have

$$rarrow \overset{\text{def}}{\leq} rarrow$$

SELF-SUB:
$$\frac{r \sqsubset t}{r \leq r}$$

AND-ELIM-R-SUB:
$$\frac{r \sqsubset t \qquad k \sqsubset t}{r \wedge k \leq r}$$

AND-ELIM-L-SUB:
$$\frac{r \sqsubset t \qquad k \sqsubset t}{r \wedge k \leq k}$$

AND-INTRO-SUB:
$$\frac{r \leq k_1 \qquad r \leq k_2}{r \leq k_1 \wedge k_2}$$

TRANS-SUB:
$$\frac{r \leq p \qquad p \leq k}{r \leq k}$$

RCON-SUB:
$$\frac{\overline{\overline{r}} \leq \overline{\overline{k}} \qquad rc \stackrel{\text{def}}{\leq} kc}{\overline{\overline{r}}\ rc \leq \overline{\overline{k}}\ kc}$$

RCON-AND-ELIM-SUB:
$$\frac{(\overline{r}_1; \overline{r}_2 \wedge \overline{r}'_2; \overline{r}_3; \overline{r}_4) \sqsubset \overline{\overline{t}}}{(\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4)\ rc \wedge (\overline{r}_1; \overline{r}'_2; \overline{r}_3; \overline{r}_4)\ rc' \leq (\overline{r}_1; \overline{r}_2 \wedge \overline{r}'_2; \overline{r}_3; \overline{r}_4)\ (rc \stackrel{\text{def}}{\wedge} rc')}$$

QUADRUPLE-SUB:
$$\frac{\overline{k}_1 \leq \overline{r}_1 \qquad \overline{r}_2 \leq \overline{k}_2 \qquad \overline{r}_3 \equiv \overline{k}_3 \qquad \overline{r}_4 \sqsubset \overline{t} \qquad \overline{k}_4 \sqsubset \overline{t}}{(\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4) \leq (\overline{k}_1; \overline{k}_2; \overline{k}_3; \overline{k}_4)}$$

VECTOR-SUB:
$$\frac{\begin{array}{c}\text{length}(\overline{r}) = \text{length}(\overline{k}) \\ \text{for } i \text{ in } 1 \ldots \text{length}(\overline{r}) \text{ we have } \overline{r}\{i\} \leq \overline{k}\{i\}\end{array}}{\overline{r} \leq \overline{k}}$$

VECTOR-EQUIV:
$$\frac{\overline{r} \leq \overline{k} \qquad \overline{k} \leq \overline{r}}{\overline{r} \equiv \overline{k}}$$

Figure 5.2: Polymorphic Subtyping Rules

and, for all nonnegative n,

$$ttuple_n \stackrel{\text{def}}{\leq} ttuple_n.$$

Theorem 2.21 (Subtypes Refine) on page 36 still holds. Lemma 2.22 (Tuple Intersection) on page 40, Fact 2.23 (Tuplesimp Sound) on page 41, Lemma 2.24 (Refinement Constructor Intersection) on page 41, and Fact 2.25 (Rconsimp Sound) on page 42 will be immediate corollaries of theorems we will prove below as part of the proof that each refinement type still has finitely many refinements. Since we only need these theorems for the type inference algorithm, we will postpone discussion of them until Section 5.5.

Fact 4.2 (Type Substitution Preserves Subtyping) on page 233 still holds, as does Fact 4.3 (Split Substitution) on page 233.

5.3 Finiteness of Refinements

Because we have made polymorphism more general, the lemmas used to prove that each ML type has only finitely many refinements become much more useful. Thus we shall describe the appropriate generalization of that proof now, before we use the lemmas in the soundness and decidability proofs.

In Section 2.9 on page 105, we created two interpretations of each refinement type. Two refinement types r and k were equivalent if and only if their interpretations $I(r)$ and $I(k)$ were equal, which was true if and only if their interpretations $i(r)$ and $i(k)$ mapped equivalent refinement types to equivalent generalized refinement types. This section preserves this property while generalizing I and i to apply to arbitrary refinement types with polymorphic type constructors. The definition of I in terms of i is straightforward, so we will discuss generalizing i.

The proper generalization of i is fairly clear once we determine what its inputs and outputs should be. Since in this chapter we have converted the "$\rightarrow$" refinement type operator from Chapter 2 into an ordinary refinement type constructor with one negative and one positive argument, reasoning by analogy with the definition of i from Chapter 2 leads one to expect that the new i will take negative type arguments for input and produce positive type arguments in its output.

It is possible to declare refinement types that behave similarly to $rarrow$, except the output of the function is represented as a refinement type constructor instead of as a positive type argument. For example, we can reuse the $pred$ datatype:

$$\texttt{datatype } (\alpha;;;)\ pred \texttt{ = Pred of } \alpha \rightarrow bool$$
$$\texttt{rectype } (\alpha;;;)\ tpred \texttt{ = Pred } (\alpha \rightarrow tt)$$
$$\texttt{and } (\alpha;;;)\ fpred \texttt{ = Pred } (\alpha \rightarrow ff)$$

In this case, it is obvious, for example, that a function f has the refinement type $tt \rightarrow tt \wedge ff \rightarrow ff$ if and only if `Pred` f has the refinement type $(tt;;;)\ tpred \wedge (ff;;;)\ fpred$. Thus we expect refinement type constructors to have an analogous role to positive type arguments: they are outputs from the interpretation.

The problem of determining the role of mixed type arguments remains. We will use this example:

$$\begin{array}{l}\texttt{datatype}\ (;;\alpha;)\ mix\ \texttt{=}\ \texttt{Mix of}\ \alpha \rightarrow (\alpha * bool) \\ \texttt{rectype}\ (;;\alpha;)\ tmix\ \texttt{=}\ \texttt{Mix}\ (\alpha \rightarrow (\alpha * tt)) \\ \quad \texttt{and}\ (;;\alpha;)\ fmix\ \texttt{=}\ \texttt{Mix}\ (\alpha \rightarrow (\alpha * ff)) \\ \quad \texttt{and}\ (;;\alpha;)\ botmix\ \texttt{=}\ \texttt{Mix}\ (\alpha \rightarrow (\alpha * \bot_{bool}))\end{array}$$

In this case, the following types are all distinct:

$$\begin{array}{c}(;;tt;)\ tmix \wedge (;;\top_{bool};)\ botmix \\ (;;tt;)\ tmix \wedge (;;\top_{bool};)\ tmix \\ (;;tt;)\ botmix \wedge (;;\top_{bool};)\ botmix \\ (;;tt;)\ botmix \wedge (;;\top_{bool};)\ tmix\end{array}$$

It seems most natural to give different interpretations to these distinct types by making mixed type arguments an input to the interpretation. From this example, it is clear that mixed arguments give rise to more distinct refinements than do negative arguments. We must therefore have more distinct interpretations of refinement types with mixed arguments; this happens because the interpretation in general is monotone for the negative arguments but not for the mixed arguments.

In Chapter 2 we had "generalized refinement types", which were either a refinement type or ns. In the argument below, we use *generalized pairs* for a similar purpose. A generalized pair is either a pair consisting of a vector of refinement types corresponding to the positive arguments of some refinement type constructor and a refinement type constructor, or it is ns. We will use the metavariables $rr?$, $kk?$, and $pp?$ to stand for generalized pairs. The operations on generalized refinement types can also be defined on generalized pairs; for example, the new definition of $\preceq$ is entirely analogous to the definition on page 106:

Definition 5.2 *We define the binary relation $\preceq$ on generalized pairs by the following cases:*

$$\begin{array}{ll}(\overline{r}; rc) \preceq (\overline{k}; kc) & \textit{if and only if } \overline{r} \leq \overline{k} \textit{ and } rc \stackrel{\text{def}}{\leq} kc \\ (\overline{r}; rc) \preceq \mathbf{ns} & \textit{always} \\ \mathbf{ns} \preceq (\overline{k}; kc) & \textit{never} \\ \mathbf{ns} \preceq \mathbf{ns}. & \end{array}$$

The definition of $\approx$ is also analogous:

Definition 5.3 *We say rr? $\approx$ kk? if rr? $\preceq$ kk? and kk? $\preceq$ rr?.*

Definition 5.4 *We define the binary operation $\triangle$ mapping pairs of generalized pairs to generalized pairs by the equations:*

$$(\overline{r}; rc) \triangle (\overline{k}; kc) = (\overline{r} \wedge \overline{k}; rc \overset{\text{def}}{\wedge} kc)$$
$$(\overline{r}; rc) \triangle \mathbf{ns} = \mathbf{ns} \triangle (\overline{r}; rc) = (\overline{r}; rc)$$
$$\mathbf{ns} \triangle \mathbf{ns} = \mathbf{ns}.$$

Definition 5.5 *Suppose s is a finite set of generalized pairs; then we define $\triangle s$ as follows:*

If s is empty, then $\triangle s = \mathbf{ns}$.

If $s = \{rr?_1, \ldots, rr?_n\}$, then $\triangle s = rr?_1 \wedge \ldots \wedge rr?_n$.

Definition 5.6 *We say $rr? \sqsubset (\overline{t}; tc)$ if $rr? = \mathbf{ns}$ or $rr? \propto (\overline{r}; rc)$ and $\overline{r} \sqsubset \overline{t}$ and $rc \overset{\text{def}}{\sqsubset} tc$.*

Definition 5.7 *If s is a finite set of generalized pairs, then we define $s \sqsubset (\overline{t}; tc)$ to mean that for all elements $rr?$ of s, we have $rr? \sqsubset (\overline{t}; tc)$.*

Fact 2.76 ($\triangle$ Elim Sub) on page 107, Fact 2.77 ($\triangle$ Intro Sub) on page 107, and Fact 2.78 (Transitivity of $\preceq$) on page 108 transplant easily to this new context, and they continue to hold.

As discussed earlier, the new interpretation of a refinement type takes as input two sequences of refinement types corresponding to the negative and mixed arguments and it outputs a generalized pair. Interpretations have a simple property that can almost be used as a definition: the interpretation i of a type r is a function f such that for all well-formed $\overline{r}$, $\overline{r}''$, and $\overline{r}'''$ of appropriate length, if there is a least pair $(\overline{r}'; rc)$ such that $r \leq (\overline{r}; \overline{r}'; \overline{r}''; \overline{r}''')rc$, then $f(\overline{r}; \overline{r}'') = (\overline{r}'; rc)$; if there is no $(\overline{r}'; rc)$ such that $r \leq (\overline{r}; \overline{r}'; \overline{r}''; \overline{r}''')rc$ then $f(\overline{r}; \overline{r}'') = \mathrm{ns}$. To avoid a circular proof, we cannot yet argue that these are all the possibilities; in principle there could be an infinite chain of distinct pairs $\ldots \leq (\overline{k}_3; kc_3) \leq (\overline{k}_2; kc_2) \leq (\overline{k}_1; kc_1)$ such that for all i we have $r \leq (\overline{r}; \overline{k}_i; \overline{r}''; \overline{r}''')kc_i$. Thus we will not use this simple property to define i; instead give a different, more constructive, definition of i and then prove that the i defined this way satisfies the simple property.

Definition 5.8 (Interpretation of a Refinement Type) *Suppose k has the form*

$$(\overline{k}_1; \overline{k}'_1; \overline{k}''_1; \overline{k}'''_1)kc_1 \wedge \ldots \wedge (\overline{k}_n; \overline{k}'_n; \overline{k}''_n; \overline{k}'''_n)kc_n$$

and suppose $k \sqsubset (\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$ and $\overline{r} \sqsubset \overline{t}$ and $\overline{r}'' \sqsubset \overline{t}''$. Then we define $i(k)(\overline{r}; \overline{r}'')$ to be

$$\triangle\{(\overline{k}'_h; kc_h) \mid h \text{ is in } 1 \ldots n \text{ and } \overline{r} \leq \overline{k}_h \text{ and } \overline{r}'' \equiv \overline{k}''_h\}.$$

We extend i to give an interpretation to generalized refinement types by defining $i(\mathbf{ns})(\overline{r};\overline{r}'') = \mathbf{ns}$.

For example, if we eliminate some of the syntactic sugar from the refinement type $tt \to ff \wedge \top_{bool} \to tt$ we get $(tt;ff;;)\ rarrow \wedge (\top_{bool};tt;)\ rarrow$. The interpretation of this is a function; if we pass the function the pair $(tt;)$ consisting of the sequence of refinement types tt with length 1 followed by the empty sequence of refinement types, the result is the generalized pair $(tt \wedge \top_{bool}; rarrow \overset{\text{def}}{\wedge} rarrow)$. This has the same information as the interpretation we had in Chapter 2.

For another example, we can compute $i((tt;;;)\ tpred \wedge (ff;;;)\ fpred)$; this is a function which, among other things, maps the pair $(tt \wedge ff;)$ to $(;tpred \overset{\text{def}}{\wedge} fpred)$.

We also give an example using the datatype mix. The refinement type $(;;tt;)\ tmix \wedge (;;\top_{bool};)\ fmix$ is not equivalent to any simpler refinement type; we have

$$i((;;tt;)\ tmix \wedge (;;\top_{bool};)\ fmix)(;tt) = (;tmix),$$

$$i((;;tt;)\ tmix \wedge (;;\top_{bool};)\ fmix)(;ff) = (;fmix),$$

and

$$i((;;tt;)\ tmix \wedge (;;\top_{bool};)\ fmix)(;\top_{bool}) = \mathbf{ns}.$$

After updating the notation, the theorems proved about i in Chapter 2 are still provable. The only real difference in the proofs is the convolutedness of the notation, so we shall omit the proofs.

The new version of Lemma 2.80 (i Monotone in Second Argument) on page 108 has two parts, because the mixed type arguments are treated differently from the negative type arguments:

Fact 5.9 ($i(k)(\overline{r};\overline{r}'')$ Monotone in $\overline{r}$) *If*

$$\overline{r}_1 \leq \overline{r}_2$$

and $\overline{r}_1 \sqsubset \overline{t}$ *and* $\overline{r}'' \sqsubset \overline{t}''$ *and* $k \sqsubset (\overline{t};\overline{t}';\overline{t}'';\overline{t}''')$*tc, then*

$$i(k)(\overline{r}_1;\overline{r}'') \preceq i(k)(\overline{r}_2;\overline{r}'').$$

Fact 5.10 ($i(k)(\overline{r};\overline{r}'')$ Respects Equivalence in $\overline{r}''$) *If*

$$\overline{r}''_1 \equiv \overline{r}''_2$$

and $\overline{r} \sqsubset \overline{t}$ *and* $\overline{r}''_1 \sqsubset \overline{t}''$ *and* $k \sqsubset (\overline{t};\overline{t}';\overline{t}'';\overline{t}''')$*tc, then*

$$i(k)(\overline{r};\overline{r}''_1) \approx i(k)(\overline{r};\overline{r}''_2).$$

Lemma 2.81 (i Monotone in First Argument) on page 109 requires little change:

Fact 5.11 (i Monotone in First Argument) *If*

$$k \leq p$$

and $\overline{r} \sqsubset \overline{t}$ *and* $\overline{r}'' \sqsubset \overline{t}''$ *and* $k \sqsubset (\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')$*tc, then*

$$i(k)(\overline{r}; \overline{r}'') \preceq i(p)(\overline{r}; \overline{r}'').$$

The corollary to Lemma 2.81 (i Monotone in First Argument) on page 109 is still simple:

Fact 5.12 (Bound on Argument to i Gives Bound on i) *If*

$$k \leq (\overline{r}; \overline{r}'; \overline{r}''; \overline{r}''')rc$$

then

$$i(k)(\overline{r}; \overline{r}'') \preceq (\overline{r}'; rc).$$

Using the facts stated above, we can prove the following generalization of Lemma 2.26 (Tuple Subtyping) on page 42 and Fact 2.28 (Refinement Constructor Subtyping) on page 45:

Corollary 5.13 (Arbitrary Constructor Subtyping) *If* $(\overline{\overline{k}})kc \leq (\overline{\overline{r}})rc$*, then* $\overline{\overline{k}} \leq \overline{\overline{r}}$ *and* $kc \overset{\text{def}}{\leq} rc$*.*

Proof: Suppose $\overline{\overline{r}} \propto \overline{r}; \overline{r}'; \overline{r}''; \overline{r}'''$ and $\overline{\overline{k}} \propto \overline{k}; \overline{k}'; \overline{k}''; \overline{k}'''$. By Fact 5.12 (Bound on Argument to i Gives Bound on i) on page 250 we have

$$i((\overline{\overline{k}})kc)(\overline{r}; \overline{r}'') \preceq (\overline{r}'; rc). \tag{5.1}$$

Thus $i((\overline{\overline{k}})kc)(\overline{r}; \overline{r}'')$ is not ns, and the definition of i gives

$$i((\overline{\overline{k}})kc)(\overline{r}; \overline{r}'') = (\overline{k}'; kc),$$

$$\overline{r} \leq \overline{k},$$

and

$$\overline{r}'' \equiv \overline{k}''.$$

From (5.1) and the definition of $\preceq$ we have

$$\overline{k}' \leq \overline{r}'$$

and

$$kc \stackrel{\text{def}}{\leq} rc. \tag{5.2}$$

Theorem 2.21 (Subtypes Refine) on page 36 holds for this system, so there is a $\overline{t}'''$ such that $\overline{k}''' \sqsubset \overline{t}'''$ and $\overline{r}''' \sqsubset \overline{t}'''$. Then the definition of $\leq$ for sequences gives $\overline{\overline{k}} \leq \overline{\overline{r}}$. This and (5.2) are our conclusion. □

We state the generalizations of the remaining lemmas and theorems from Section 2.8 so a determined reader will be able to reconstruct the details of the reasoning without going astray. Nothing very interesting is happening here, so other readers can skip to the next section.

Fact 5.14 (i Gives an Upper Bound) *If*

$$i(k)(\overline{r}; \overline{r}'') \preceq (\overline{r}'; rc)$$

and $k \sqsubset (\overline{r}; \overline{r}'; \overline{r}''; \overline{r}''')tc$ *and* $\overline{r}''' \sqsubset \overline{t}'''$, *then*

$$k \leq (\overline{r}; \overline{r}'; \overline{r}''; \overline{r}''')rc.$$

Fact 5.15 (Ordering on i) *If, for all* $\overline{k} \sqsubset \overline{t}$ *and all* $\overline{k}'' \sqsubset \overline{t}''$, *we have*

$$i(r)(\overline{k}; \overline{k}'') \preceq i(p)(\overline{k}; \overline{k}''),$$

and $r \sqsubset (\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$ *and* $p \sqsubset (\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$, *then*

$$r \leq p.$$

Fact 5.16 (i Preserves Information) *Suppose that* r_1 *and* r_2 *both refine* $(\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$. *Then*

for all $\overline{k}_1 \sqsubset \overline{t}$ *and* $\overline{k}_2 \sqsubset \overline{t}$ *and*
$\overline{k}_1'' \sqsubset \overline{t}''$ *and* $\overline{k}_2'' \sqsubset \overline{t}''$
we have
$(\overline{k}_1 \equiv \overline{k}_2)$ *and* $(\overline{k}_1'' \equiv \overline{k}_2'')$ *imply*
$(i(r_1)(\overline{k}_1; \overline{k}_1'') \approx i(r_2)(\overline{k}_2; \overline{k}_2''))$

if and only if

$$r_1 \equiv r_2.$$

Definition 5.17 *We define the equivalence class of a generalized refinement type* $r?$ *(written* $C(r?)$*) to be the set* $\{r?' \mid r?' \approx r?\}$.

We define the equivalence class of a generalized pair rr? *(written* $C(rr?)$*) to be the set* $\{rr?' \mid rr?' \approx rr?'\}$.

We define the equivalence class of a sequence of refinement types $\overline{r}$ *(written* $C(\overline{r})$*) to be the set* $\{\overline{r}' \mid \overline{r}' \equiv \overline{r}\}$.

Definition 5.18 *We define the set of equivalence classes of refinements of an ML type t (written EC(t)) to be $\{C(r?) \mid r? \sqsubset t\}$.*

We define the set of equivalence classes of refinements of a pair $(\overline{t}; tc)$ (written EC$(\overline{t}; tc)$) to be $\{C(rr?) \mid rr? \sqsubset (\overline{t}; tc)\}$.

We define the set of equivalence classes of refinements of a sequence $\overline{t}$ (written EC$(\overline{t})$) to be $\{C(\overline{r}) \mid \overline{r} \sqsubset \overline{t}\}$.

We shall use c as a metavariable standing for the equivalence class of some refinement type, $c?$ as a metavariable standing for the equivalence class of some generalized refinement type, $cc?$ as a metavariable standing for the equivalence class of some generalized pair, and $\overline{c}$ as a metavariable standing for the equivalence class of some sequence.

Definition 5.19 *If $r \sqsubset (\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$ and $cc? \in$ EC$(\overline{t}'; tc)$ and $\overline{c} \in$ EC$(\overline{t})$ and $\overline{c}'' \in$ EC$(\overline{t}'')$ then we write*

$$cc? = I(r)(\overline{c}; \overline{c}'')$$

if there is a $\overline{k}$ and a $\overline{k}''$ in $\overline{c}''$ such that

$$cc? = C(i(r)(\overline{k}; \overline{k}'')).$$

By Fact 5.9 ($i(k)(\overline{r}; \overline{r}'')$ Monotone in $\overline{r}$) on page 249 and Fact 5.10 ($i(k)(\overline{r}; \overline{r}'')$ Respects Equivalence in $\overline{r}''$) on page 249, for all r we know that $I(r)$ is a function.

Fact 5.20 (I Preserves Equivalence) *If r and r' refine $(\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$ then*

$$r \equiv r'$$

if and only if

$$\textit{for all } \overline{c} \in \text{EC}(\overline{t}) \textit{ and all } \overline{c}'' \in \text{EC}(\overline{t}'') \textit{ we have } I(r)(\overline{c}; \overline{c}'') = I(r')(\overline{c}; \overline{c}'').$$

And finally,

Fact 5.21 (Finite Refinements) *For each ML type u we have EC(u) is finite.*

and, once we define type inference, the same simple argument for principal types used in Chapter 2 will hold:

Fact 5.22 (Principal Refinement Types) *If*

$$\text{VR} \vdash e : r$$

then there is a k such that

$$\text{VR} \vdash e : k$$

and for all p we have

$$\text{VR} \vdash e : p \textit{ implies } k \leq p.$$

5.4 Splitting

Splitting does not simplify as much as subtyping did. We only need a syntactic change to make the RCON-SPLIT rule in Figure 2.5 on page 48 accommodate polymorphic constructors:

$$\text{RCON-SPLIT: } \frac{rc \overset{\text{def}}{\asymp} sc}{(\overline{\overline{r}})rc \asymp \{(\overline{\overline{r}})kc \mid kc \in sc\}}$$

None of the other splitting rules in the system change; in particular, the TUPLE-SPLIT rule does not change. It is tempting to "generalize" it to get an incorrect rule for splitting polymorphic types where all the arguments are positive:

$$\text{BOGUS: } \frac{k_i \asymp s}{(;;k_1,\ldots,k_{i-1},k_i,k_{i+1},\ldots,k_m;)rc \asymp \{(;;k_1,\ldots,k_{i-1},p,k_{i+1},\ldots,k_m;)rc \mid p \in s\}}$$

This incorrect rule would allow us to use $\top_{bool} \asymp \{tt, ff\}$ to derive $\top_{bool}\ ev \asymp \{tt\ ev, ff\ ev\}$. This is not sound; for example, the value `cons (true, cons (false, nil))` has the type $\top_{bool}\ ev$ but it does not have either of the types $tt\ ev$ or $ff\ ev$.

To make type inference tractable, we assume that if a refinement type constructor splits, it has no negative or mixed type arguments. Formally,

Assumption 5.23 (Split Positive) *If* $rc \overset{\text{def}}{\asymp} \{rc_1,\ldots,rc_n\}$, *then* arity($rc$) *has the form* $(0;x;0;y)$ *for some nonnegative integers* x *and* y.

Without this assumption, there might be splittable refinement types that can only be expressed as an intersection of other refinement types; for example, we could have the declaration

```
datatype (α;;;) doublepred = Pred1 of α → bool | Pred2 of α → bool
rectype (α;;;) tpred12 = Pred1 of α → tt | Pred2 of α → tt
    and (α;;;) tpred1 = Pred1 of α → tt
    and (α;;;) tpred2 = Pred2 of α → tt
    and (α;;;) fpred12 = Pred1 of α → ff | Pred2 of α → ff
```

where $tpred12 \overset{\text{def}}{\asymp} \{tpred1, tpred2\}$. Then we might have to determine that the principal split of the refinement type $(tt;;;)\ tpred12 \wedge (ff;;;)\ fpred12$ is

$$\{(tt;;;)\ tpred1 \wedge (ff;;;)\ fpred12, (tt;;;)\ tpred2 \wedge (ff;;;)\ fpred12\}.$$

In general, it seem that we might have to search over all of the supertypes of the type we start with to find splits, and then combine these to get the principal split. The assumption

above saves us from that; with the assumption, all splittable types are equivalent to a type of the form $(\overline{\overline{r}})rc$ where rc has a predefined split.

The theorems Theorem 2.31 (Splits Are Subtypes I) on page 49, Corollary 2.32 (Split Types Refine I) on page 51, Fact 2.33 (Splits Are Subtypes II) on page 51, and Fact 2.34 (Split Types Refine II) on page 51 continue to hold for the new system. To get Fact 2.35 (Splits of Arrows are Simple) on page 51 to hold in the new system, we need to assume that $rarrow$ has no interesting splits:

Assumption 5.24 (Arrow Does Not Split) *There is no sc such that rarrow* $\overset{\text{def}}{\asymp}$ *sc.*

Fact 2.37 (Splits are Nonempty) on page 51 continues to hold, as do Lemma 2.43 (Split Intersection) on page 54, Lemma 2.45 (Principal Split Implies Useless Splitting Fragments) on page 58, and Lemma 2.46 (Fragments of Principal Split have Useless Splits) on page 58.

5.5 Refinement Type Inference

The assumptions we make about value constructors now have type variables embedded. To say that a constructor c maps values with type r to values with type $(\overline{\overline{\alpha}})rc$, we write

$$c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc.$$

This notation implies that c maps all instances of r to the corresponding instances of $(\overline{\overline{\alpha}})rc$. Only two refinement type inference rules appearing in Figure 2.6 on page 60 and updated on page 4.2 have to change to accommodate this:

CONSTR-TYPE:

$$\frac{c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc \quad \text{VR} \vdash e : [\overline{\overline{r}}/\overline{\overline{\alpha}}]r \quad \overline{\overline{r}} \sqsubset \overline{\overline{t}}}{\text{VR} \vdash c[\overline{\overline{t}}]\ \ e : (\overline{\overline{r}})rc}$$

CASE-TYPE:

$$\frac{\begin{array}{c}\text{VR} \vdash e_0 : (\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_m)rc_m \\ r \sqsubset u \\ \text{rtom(VR)} \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}u) :: u \\ \text{for all } i \text{ in } 1 \ldots n \text{ and all } r_1, \ldots, r_m, \text{ whenever} \\ \quad \text{for all } j \text{ in } 1 \ldots m \text{ we have } c_i \overset{\text{def}}{:} r_j \hookrightarrow (\overline{\overline{\alpha}})rc_j \\ \text{we have} \\ \quad \text{VR} \vdash e_i : ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m) \rightarrow r\end{array}}{\text{VR} \vdash (\texttt{case}\ e_0\ \texttt{of}\ c_1\ \texttt{=>}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ c_n\ \texttt{=>}\ e_n\ \texttt{end:}u) : r}$$

The CONSTR-TYPE rule is fairly intuitive. To infer that a value constructor returns a polymorphic type $(\overline{\overline{r}})rc$, we check that $\overline{\overline{r}}$ is well-formed, that the value constructor maps

some type r to rc, and that the argument of the value constructor has an appropriate instance of r as its type. For example, we can use this rule to conclude that the expression `cons`$[bool]$ (`true`[] (), `nil`$[bool]$ ()) has the type $(;tt;;)\ od$, assuming we have the premises

$$\texttt{cons} \stackrel{\text{def}}{:} (\alpha * (;\alpha;;)\ ev) \hookrightarrow (;\alpha;;)\ od,$$

$$\cdot \vdash (\texttt{true}[]\ (),\ \texttt{nil}[bool]\ ()) : tt * (;tt;;)\ ev,$$

and

$$tt \sqsubset bool.$$

The CASE-TYPE rule is somewhat more complex; the difficult part is the premise beginning with "for all i in $1 \ldots n$...". An intuitive reasonable reading of this complex premise of CASE-TYPE is

> for all branches of the `case` statement and all inputs to the constructor in that branch, if
> giving an input to the constructor for this branch yields the type of the case object
> then
> giving that input to the type of this branch must yield the type of the `case` statement.

We can translate this into the formal notation used in the inference rule as follows:

- "For all branches of the `case` statement" becomes "for all i in $1 \ldots n$".
- "The case object" is "e_0".
- "The type of the case object" is $(\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_m)rc_m$.
- "The constructor in that branch" is "c_i".
- "For all inputs to the constructor in that branch" becomes "for all $r_1, \ldots, r_m$". Roughly speaking, the input to the constructor is $[\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m$.
- "Giving an input to the constructor of this branch yields the type of the case object" translates approximately to

  "$c_i : ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m) \rightarrow ((\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_m)rc_m)$".

  However, this is not well formed, since constructors by themselves are not expressions. Doing this one component at a time yields the still ill-formed translation

  "for all j in $1 \ldots m$ we have $c_i : [\overline{\overline{r}}_j/\overline{\overline{\alpha}}]r_j \rightarrow (\overline{\overline{r}}_m)rc_m$".

  We can make a well-formed translation without changing the meaning in any important way by omitting the instantiation in the type of c_i; this yields

  "for all j in $1 \ldots m$ we have $c_i \stackrel{\text{def}}{:} r_j \hookrightarrow (\overline{\overline{\alpha}})rc_j$".

- "The type of the `case` statement" is simply "r".
- "This branch" is "e_i".
- "Giving that input to the type of this branch must yield the type of the `case` statement" translates to "VR $\vdash e_i : ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m) \rightarrow r$".

For example, if we use the *pred* datatype introduced on page 240, then the expression

$$\begin{array}{l}\texttt{case Pred (fn x:}\mathit{bool}\texttt{ => x[]) of}\\ \quad\texttt{Pred => fn f:}\mathit{bool}\rightarrow\mathit{bool}\texttt{ => f[]}\\ \texttt{end:}\mathit{bool}\rightarrow\mathit{bool}\end{array}$$

has the type $tt \rightarrow tt \wedge \mathit{ff} \rightarrow \mathit{ff}$ because the following premises hold:

$$\cdot \vdash \texttt{Pred (fn x:}\mathit{bool}\texttt{ => x[])} : (tt;;;)\ \mathit{tpred} \wedge (\mathit{ff};;;)\ \mathit{fpred},$$

$$tt \rightarrow tt \wedge \mathit{ff} \rightarrow \mathit{ff} \sqsubset \mathit{bool} \rightarrow \mathit{bool},$$

$$\cdot \vdash (\texttt{case ... end:}\mathit{bool} \rightarrow \mathit{bool}) :: \mathit{bool} \rightarrow \mathit{bool},$$

and

$$\begin{array}{l}\texttt{Pred} \overset{\text{def}}{:} (\alpha \rightarrow tt) \hookrightarrow (\alpha;;;)\ \mathit{tpred} \text{ and}\\ \texttt{Pred} \overset{\text{def}}{:} (\alpha \rightarrow \mathit{ff}) \hookrightarrow (\alpha;;;)\ \mathit{fpred} \text{ imply}\\ \cdot \vdash \texttt{fn f:}\mathit{bool}\rightarrow\mathit{bool}\texttt{ => f[]} : tt \rightarrow tt \wedge \mathit{ff} \rightarrow \mathit{ff}\end{array}\ .$$

5.5.1 Positions of Type Variables

We need to make some assumptions about how the type variables appear in the $\overset{\text{def}}{:}$ relation. First we must assume that the type variables appear in the proper position; for example, if α is in the positive position of $\overline{\overline{\alpha}}$, and $c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc$, then $[k/\alpha]r$ should become larger as k becomes larger. We will give a formal definition of this assumption below, but first we will give a concrete example where refinement type inference is unsound if the assumption does not hold.

We will assume a datatype $(\alpha;;;)\ box$ with one negative type argument, one refinement $\top_{box}$, and one value constructor `Box`. We shall assume the constructor `Box` has these behaviors:

$$\texttt{Box} \overset{\text{def}}{::} \alpha \hookrightarrow (\alpha;;;)\ box$$

$$\texttt{Box} \overset{\text{def}}{:} \alpha \hookrightarrow (\alpha;;;)\ \top_{box}$$

Since α occupies the first position in the quadruple $(\alpha;;;)$ and the first position is negative, this behavior for `Box` violates the assumption we are discussing: as a type substituted for α gets larger, the type α on the left hand side of the $\hookrightarrow$ also gets larger, but the type $(\alpha;;;)\ \top_{box}$ on the right side gets smaller.

We will also use the value constructor `true` for the booleans. If we do not use any of the convenient abbreviations we have introduced so far, `true` has these behaviors:

$$\texttt{true} \overset{\text{def}}{::} (;;;)\ ttuple_0 \hookrightarrow (;;;)\ bool$$

$$\texttt{true} \overset{\text{def}}{:} (;;;)\ rtuple_0 \hookrightarrow (;;;)\ tt$$

Using the abbreviations, we can make these look more familiar:

$$\texttt{true} \overset{\text{def}}{::} tunit \hookrightarrow bool$$

$$\texttt{true} \overset{\text{def}}{:} runit \hookrightarrow tt$$

We will also use the assumption that $\bot_{bool} \overset{\text{def}}{\leq} tt$.

In this context, we can now show that the expression

```
case Box[bool;;;] (true ()) of
   Box => fn x:bool => x[]
end:bool
```

has the refinement type $\bot_{bool}$. Since this expression evaluates to `true ()`, and `true ()` does not have the type $\bot_{bool}$, our type system is not sound with the assumptions we have made so far.

First we find a type for `Box`$[tt;;;]$ `(true ())`. The expression `true ()` obviously has the type tt. By CONSTR-TYPE and the assumed behavior of `Box`, the expression

$$\texttt{Box}[tt;;;]\ (\texttt{true}\ ())$$

then has the type $(tt;;;)\ \top_{box}$. Now we can do the step where we lose soundness: since $\bot_{bool} \leq tt$, by RCON-SUB we have $(tt;;;)\ \top_{box} \leq (\bot_{bool};;;)\ \top_{box}$; thus by WEAKEN-TYPE we can infer that `Box`$[tt;;;]$ `(true ())` has the type $(\bot_{bool};;;)\ \top_{box}$.

Now we can continue to find a type for the `case` statement as a whole. This is a use of CASE-TYPE with the premises

$$\cdot \vdash \texttt{Box}[tt;;;]\ (\texttt{true}\ ()) : (\bot_{bool};;;)\ \top_{box},$$

$$\bot_{bool} \sqsubset bool,$$

$$\cdot \vdash \texttt{case}\ \ldots\ \texttt{end}:bool :: bool,$$

and

$$\texttt{Box} \overset{\text{def}}{:} \alpha \rightarrow (\alpha;;;)\ \top_{box} \text{ and } \cdot \vdash \texttt{fn x}:bool\ \texttt{=>}\ \texttt{x}[] : \bot_{bool} \rightarrow \bot_{bool}.$$

The conclusion of CASE-TYPE is $\cdot \vdash (\texttt{case}\ \ldots\ \texttt{end}:bool) : tt$. As discussed above, if this expression has this type, then type inference is not sound.

To prevent this, we need to define what it means for a type variable to be positive, negative, mixed, or absent from a refinement type, and we will require that whenever $c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc$, the positive type arguments of rc are positive in r, the negative ones are negative, and so forth. The definition is somewhat wordy, but very regular.

Definition 5.25 (Negative, Positive, Ignored) *We say a type variable α is negative (or positive or ignored) in a list of refinement types $\overline{r}$ if α is negative (or positive or ignored, respectively) in each element of $\overline{r}$.*

A type variable α is negative *in a refinement type r if*

- $r \propto r_1 \wedge r_2$ *and α is negative in both r_1 and r_2, or*
- $r \propto \beta$ *where $\beta \neq \alpha$, or*
- $r \propto (\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4)rc$ *and α is positive in $\overline{r}_1$, negative in $\overline{r}_2$, and ignored in $\overline{r}_3$.*

A type variable α is positive *in a refinement type r if*

- $r \propto r_1 \wedge r_2$ *and α is positive in r_1 and r_2, or*
- $r \propto \beta$ *(whether or not $\alpha = \beta$), or*
- $r \propto (\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4)rc$ *and α is negative in $\overline{r}_1$, positive in $\overline{r}_2$, and is ignored in $\overline{r}_3$.*

A type variable α is ignored *in a refinement type r if*

- $r \propto r_1 \wedge r_2$ *and α is ignored in both r_1 and r_2, or*
- $r \propto \beta$ *where $\alpha \neq \beta$*
- $r \propto (\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4)rc$ *and α is ignored in $\overline{r}_1$, $\overline{r}_2$, and $\overline{r}_3$. (It does not matter whether it appears in $\overline{r}_4$.)*

Definition 5.26 (Varies properly) *We say that a quadruple of type variables $\overline{\alpha}_1; \overline{\alpha}_2; \overline{\alpha}_3; \overline{\alpha}_4$* varies properly *in a refinement type r if all the variables in $\overline{\alpha}_1$ are negative in r, all variables in $\overline{\alpha}_2$ are positive in r, and all variables in $\overline{\alpha}_4$ are ignored in r.*

Assumption 5.27 (Variance) *If $c \stackrel{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc$, then $\overline{\overline{\alpha}}$ varies properly in r.*

Fact 5.28 (Variant Weakening) *If $\overline{\overline{\alpha}}$ varies properly in r, and $\overline{\overline{r}} \leq \overline{\overline{r}}'$, and $r \sqsubset t$, then*

$$[\overline{\overline{r}}/\overline{\overline{\alpha}}]r \leq [\overline{\overline{r}}'/\overline{\overline{\alpha}}]r.$$

The proof is by induction on r.

5.5.2 Intersection and Polymorphism

In this Subsection we will make an assumption that eliminates the need to use RCON-AND-ELIM-SUB to reason about types for constructors. The assumption is:

Assumption 5.29 (Predefined Intersection Distributivity) *For all value constructors c, if*

$$c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc$$

and

$$c \overset{\text{def}}{:} r' \hookrightarrow (\overline{\overline{\alpha}})rc'$$

and $\overline{\overline{\alpha}}$ has the form $(\overline{\alpha}_1;\overline{\alpha}_2;\overline{\alpha}_3;\overline{\alpha}_4)$, then for any well-formed $\overline{k}$ and $\overline{k}'$ of appropriate length, we have

$$[\overline{k}/\overline{\alpha}_2]r \wedge [\overline{k}'/\overline{\alpha}_2]r' \leq [\overline{k} \wedge \overline{k}'/\overline{\alpha}_2](r \wedge r').$$

Now we can explain which uses of RCON-AND-ELIM-SUB this makes unnecessary. Suppose an expression e has the type $[\overline{k}/\overline{\alpha}_2]r \wedge [\overline{k}'/\overline{\alpha}_2]r'$. By WEAKEN-TYPE it has each of the types $[\overline{k}/\overline{\alpha}_2]r$ and $[\overline{k}'/\overline{\alpha}_2]r'$. Then we can use CONSTR-TYPE twice to determine, for an appropriate $\overline{\overline{t}}$, that $c[\overline{\overline{t}}]\ \ e$ has each of the types $(\overline{\alpha}_1;\overline{k};\overline{\alpha}_3;\overline{\alpha}_4)rc$ and $(\overline{\alpha}_1;\overline{k}';\overline{\alpha}_3;\overline{\alpha}_4)rc'$. Then AND-INTRO-TYPE tells us it has the type $(\overline{\alpha}_1;\overline{k};\overline{\alpha}_3;\overline{\alpha}_4)rc \wedge (\overline{\alpha}_1;\overline{k}';\overline{\alpha}_3;\overline{\alpha}_4)rc'$, and WEAKEN-TYPE and RCON-AND-ELIM-SUB tells us it has the type $(\overline{\alpha}_1;\overline{k}\wedge\overline{k}';\overline{\alpha}_3;\overline{\alpha}_4)(rc \overset{\text{def}}{\wedge} rc')$.

Predefined Intersection Distributivity tells us we can come to the same conclusion without using RCON-AND-ELIM-SUB. First we use WEAKEN-TYPE to conclude that e has the type $[\overline{k} \wedge \overline{k}'/\overline{\alpha}_2](r \wedge r')$; then by Assumption 2.52 (Constructor Argument Strengthen) on page 67 and Assumption 2.51 (Constructor And Introduction) on page 67 we have $c \overset{\text{def}}{:} r \wedge r' \hookrightarrow (\overline{\overline{\alpha}})(rc \overset{\text{def}}{\wedge} rc')$, and then by CONSTR-TYPE we know that $c[\overline{\overline{t}}]\ \ e$ has the type

$$(\overline{\alpha}_1;\overline{k} \wedge \overline{k}';\overline{\alpha}_3;\overline{\alpha}_4)(rc \overset{\text{def}}{\wedge} rc').$$

Having an assumption that makes it unnecessary to use certain inference rules with constructors is something we have done before. For example, Assumption 2.51 (Constructor And Introduction) on page 67 makes it unnecessary to use AND-INTRO-TYPE in some cases, and Assumption 2.52 (Constructor Argument Strengthen) on page 67 and Assumption 2.53 (Constructor Result Weaken) on page 67 make some uses of WEAKEN-TYPE unnecessary. All these assumptions eliminate the need to use refinement type inference to infer types for constructors from the CASE-TYPE rule. Perhaps it would be possible to make a system with fewer assumptions but a more complex proof if we used refinement type inference for the constructors in `case` statements; but that is beyond the scope of this thesis.

There are no interesting changes to the theorems asserting compatibility between refinement type inference and ML type inference.

5.6 Soundness

The statement and proof of Lemma 4.4 (Environment Modification) on page 235 do not change. The new version of Lemma 2.67 (Piecewise Intersection) on page 84 needs to make nontrivial use of Predefined Intersection Distributivity; we will restate the lemma and show how it depends on Predefined Intersection Distributivity before we show the modifications to the proof.

Lemma 5.30 (Piecewise Intersection) *If*

$$\textit{for all } i \textit{ in } 1 \ldots m \textit{ we have } \cdot \Vdash v : (\overline{\overline{k}}_i)\mathit{kc}_i \tag{5.3}$$

and

$$(\overline{\overline{k}}_1)\mathit{kc}_1 \wedge \ldots \wedge (\overline{\overline{k}}_m)\mathit{kc}_m \leq (\overline{\overline{r}}_1)\mathit{rc}_1 \wedge \ldots \wedge (\overline{\overline{r}}_n)\mathit{rc}_n \tag{5.4}$$

then for all j *in* $1 \ldots n$ *we have*

$$\cdot \Vdash v : (\overline{\overline{r}}_j)\mathit{rc}_j.$$

Without Predefined Intersection Distributivity, this is false. For a counterexample, suppose we have the declarations

$$\texttt{datatype } (\alpha;;;)\ d \texttt{ = C of } \mathit{bool} \to \alpha$$
$$\texttt{rectype } (\alpha;;;)\ z \texttt{ = C } (\mathit{tt} \to \alpha) \texttt{ | C } (\mathit{ff} \to \alpha)$$

Then, if it were not for Predefined Intersection Distributivity, we could use a straightforward procedure to determine that C has the behaviors

$$C \overset{\text{def}}{:} (\mathit{tt} \to \alpha) \hookrightarrow (\alpha;;;)\ z$$

and

$$C \overset{\text{def}}{:} (\mathit{ff} \to \alpha) \hookrightarrow (\alpha;;;)\ z$$

but not

$$C \overset{\text{def}}{:} (\bot_{\mathit{bool}} \to \alpha) \hookrightarrow (\alpha;;;)\ z.$$

Then these premises of Piecewise Intersection could be true:

$$\cdot \Vdash \texttt{C}[\mathit{bool}]\ (\texttt{fn x}:\mathit{bool}\ \texttt{=>}\ \texttt{x}[]) : (\mathit{tt};;;)\ z$$

$$\cdot \Vdash \texttt{C}[\mathit{bool}]\ (\texttt{fn x}:\mathit{bool}\ \texttt{=>}\ \texttt{x}[]) : (\mathit{ff};;;)\ z$$

$$(\mathit{tt};;;)\ z \wedge (\mathit{ff};;;)\ z \leq (\bot_{\mathit{bool}};;;)\ z$$

but we would not have the conclusion

$$\cdot \Vdash \texttt{C}[\mathit{bool}]\ (\texttt{fn x}:\mathit{bool}\ \texttt{=>}\ \texttt{x}[]) : (\bot_{\mathit{bool}};;;)\ z$$

because the only way to derive this conclusion uses WEAKEN-TYPE as the last inference, and the meaning of $\Vdash$ specifically excludes WEAKEN-TYPE as the last inference.

Proof: By induction on the derivation of $(\overline{\overline{k}}_1)kc_1 \wedge \ldots \wedge (\overline{\overline{k}}_m)kc_m \leq (\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_n)rc_n$.

Case: RCON-SUB. Then $m = n = 1$ and the premises of RCON-SUB are $\overline{\overline{k}}_1 \leq \overline{\overline{r}}_1$ and $kc_1 \overset{\text{def}}{\leq} rc_1$. From here we take cases on the form of v.

SubCase: $v \propto c[tl]\ v'$. The last inference of $\cdot \Vdash v : (\overline{\overline{k}}_1)kc_1$ must be CONSTR-TYPE with the premises

$$c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})kc_1,$$

$$\cdot \vdash v' : [\overline{\overline{k}}_1/\overline{\overline{\alpha}}]r,$$

and

$$\overline{\overline{k}}_1 \sqsubset \overline{\overline{t}}.$$

By Assumption 2.53 (Constructor Result Weaken) on page 67 and $kc_1 \overset{\text{def}}{\leq} rc_1$, we have

$$c \overset{\text{def}}{:} r \hookrightarrow (\overline{\overline{\alpha}})rc_1. \tag{5.5}$$

By Assumption 5.27 (Variance) on page 258 we know that $\overline{\overline{\alpha}}$ varies properly in r. Thus Fact 5.28 (Variant Weakening) on page 258 gives $[\overline{\overline{k}}_1/\overline{\overline{\alpha}}]r \leq [\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r$, and then WEAKEN-TYPE gives

$$\cdot \vdash v' : [\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r$$

and CONSTR-TYPE and (5.5) give

$$\cdot \vdash c[\overline{\overline{t}}]\ v' : (\overline{\overline{r}}_1)rc_1,$$

which is our conclusion.

SubCase: Otherwise. Omitted.

Case: RCON-AND-ELIM-SUB. Then (5.4) has the form

$$(\overline{k}_1; \overline{k}_2; \overline{k}_3; \overline{k}_4)kc_1 \wedge (\overline{k}_1; \overline{k}'_2; \overline{k}_3; \overline{k}_4)kc_2 \leq (\overline{k}_1; \overline{k}_2 \wedge \overline{k}'_2; \overline{k}_3; \overline{k}_4)(kc_1 \overset{\text{def}}{\wedge} kc_2).$$

From here we shall take cases on the form of v.

SubCase: $v \propto c[\overline{\overline{t}}]\ v'$. Then (5.3) says

$$\cdot \Vdash c[\overline{\overline{t}}]\ v' : (\overline{\overline{k}}_1)kc_1$$

and

$$\cdot \Vdash c[\overline{\overline{t}}]\ \ v' : (\overline{\overline{k}}_2)kc_2.$$

The last inferences of each of these must be CONSTR-TYPE with the premises

$$\begin{array}{c} c \stackrel{\text{def}}{:} k_1 \hookrightarrow (\overline{\overline{\alpha}})kc_1 \\ \cdot \vdash v' : [\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \\ \overline{\overline{k}}_1 \sqsubset \overline{\overline{t}} \\ c \stackrel{\text{def}}{:} k_2 \hookrightarrow (\overline{\overline{\alpha}})kc_2 \\ \cdot \vdash v' : [\overline{\overline{k}}_2/\overline{\overline{\alpha}}]k_2 \\ \overline{\overline{k}}_2 \sqsubset \overline{\overline{t}} \end{array}$$

Then AND-INTRO-TYPE gives

$$\cdot \vdash v' : [\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge [\overline{\overline{k}}_2/\overline{\overline{\alpha}}]k_2.$$

Suppose $\overline{\overline{\alpha}}$ has the form $\overline{\alpha}_1;\overline{\alpha}_2;\overline{\alpha}_3;\overline{\alpha}_4$; then Assumption 5.29 (Predefined Intersection Distributivity) on page 259 gives

$$[\overline{k}_2/\overline{\alpha}_2]k_1 \wedge [\overline{k}'_2/\overline{\alpha}_2]k_2 \leq [\overline{k}_2 \wedge \overline{k}'_2/\overline{\alpha}_2](k_1 \wedge k_2).$$

Using Fact 4.2 (Type Substitution Preserves Subtyping) on page 233 on this gives

$$[\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge [\overline{\overline{k}}_2/\overline{\overline{\alpha}}]k_2 \leq [\overline{\overline{r}}_1/\overline{\overline{\alpha}}](k_1 \wedge k_2).$$

Then WEAKEN-TYPE gives

$$\cdot \vdash v' : [\overline{\overline{r}}_1/\overline{\overline{\alpha}}](k_1 \wedge k_2).$$

Assumption 2.18 (and-intro-$\stackrel{\text{def}}{\leq}$) on page 34 and Assumption 2.52 (Constructor Argument Strengthen) on page 67 give

$$c \stackrel{\text{def}}{:} (k_1 \wedge k_2) \hookrightarrow (\overline{\overline{\alpha}})(kc_1 \stackrel{\text{def}}{\wedge} kc_2),$$

and then CONSTR-TYPE gives

$$\cdot \vdash c[tl]\ \ v' : (\overline{\overline{r}})(kc_1 \stackrel{\text{def}}{\wedge} kc_2),$$

which is our conclusion.

SubCase: Otherwise. Omitted.

Case: Otherwise. Omitted. □

Lemma 2.68 (Subtype Irrelevancy) on page 88, Theorem 2.69 (Splitting Value Types) on page 89, and Fact 4.5 (Refinement Type Substitution) on page 236 continue to hold with no interesting changes. The version of Value Substitution as revised in Chapter 4 on page 236 also has only notational changes. Refinement Type Soundness as revised in Chapter 4 on page 237 has no interesting changes either; the only nontrivial changes needed to deal with polymorphic type constructors are in the lemmas, not in the main theorem.

5.7 Decidability

The changes in this chapter only affect the cases of the type inference algorithm that deal with constructors and `case` statements. The changes required to the algorithms are intuitive; the changes required to the proofs are simple, except the proof that the algorithm derives principal types for `case` statements is awkward.

Since types have become more general, we need to generalize some of the utility functions used by `infer`. First, we make versions of `allrefs` that find refinements for a vector or quadruple of refinement types:

$$
\begin{array}{l}
\texttt{fun vallrefs } \overline{t} \texttt{ =} \\
\quad \{\overline{r} \mid \texttt{length } \overline{r} = \texttt{length } \overline{t} \text{ and} \\
\qquad \text{for } i \in 1 \ldots \texttt{length } \overline{t} \text{ we have} \\
\qquad \overline{r}\{i\} \in \texttt{allrefs } \overline{t}\{i\}\} \\
\texttt{fun qallrefs } \overline{t}_1; \overline{t}_2; \overline{t}_3; \overline{t}_4 \texttt{ =} \\
\quad \{\overline{r}_1; \overline{r}_2; \overline{r}_3; \overline{r}_4 \mid \\
\qquad \overline{r}_1 \in \texttt{vallrefs } \overline{t}_1 \text{ and} \\
\qquad \overline{r}_2 \in \texttt{vallrefs } \overline{t}_2 \text{ and} \\
\qquad \overline{r}_3 \in \texttt{vallrefs } \overline{t}_3 \text{ and} \\
\qquad \overline{r}_4 \in \texttt{vallrefs } \overline{t}_4\}
\end{array}
$$

The new definition of the interpretation i works for arbitrary refinement types, instead of just functions. We call the function for computing this `iconstr` by analogy with the `ifn` function defined on page 120. In this definition, `iconstr` $r?$ $(\overline{p}; \overline{p}'')$ $(\overline{t}; \overline{t}'')$ computes $i(r?)(\overline{p}; \overline{p}'')$, assuming that $\overline{p} \sqsubset \overline{t}$ and $\overline{p}'' \sqsubset \overline{t}''$ and, for some tc, $\overline{t}'$, and $\overline{t}'''$ we have $r? \sqsubset (\overline{t}; \overline{t}'; \overline{t}''; \overline{t}''')tc$. This definition assumes that $\triangle$`fn` has been revised to work on generalized pairs, and that `vsubtypep` is a generalization of `subtypep` that works on vectors; both of these are easy to write.

$$
\begin{array}{l}
\texttt{fun iconstr } r? \ (\overline{p}; \overline{p}'') \ (\overline{t}; \overline{t}'') \texttt{ =} \\
\quad \texttt{if } r? \texttt{ = ns then ns} \\
\quad \texttt{else} \\
\qquad \texttt{let val } (\overline{r}_1; \overline{r}'_1; \overline{r}''_1; \overline{r}'''_1)rc_1 \wedge \ldots \wedge (\overline{r}_n; \overline{r}'_n; \overline{r}''_n; \overline{r}'''_n)rc_n \texttt{ = } r? \\
\qquad \texttt{in} \\
\qquad\quad \triangle\texttt{fn } \{(\overline{r}'_h; rc_h) \mid \\
\qquad\qquad\qquad h \in 1 \ldots n \text{ and } \texttt{vsubtypep } \overline{p} \ \overline{r}_h \ \overline{t} \text{ and} \\
\qquad\qquad\qquad \texttt{vsubtypep } \overline{p}'' \ \overline{r}''_h \ \overline{t}'' \text{ and } \texttt{vsubtypep } \overline{r}''_h \ \overline{p}'' \ \overline{t}''\} \\
\qquad \texttt{end}
\end{array}
$$

It is easy to give an alternative definition of the old function `ifn` in terms of `iconstr`. Here `ifn` $r?$ p t evaluates to $i(r?)(p)$, using the old definition of i from Chapter 2, assuming that $p \sqsubset t$ and for some u we have $r? \sqsubset t \to u$.

```
fun ifn r? p t =
   case iconstr r? (p;) (t;) of
      ns => ns
    | (k; rarrow) => k
```

We also define a utility function `deval` that de-evaluates a constructor; more specifically, if $c \stackrel{\text{def}}{::} r \hookrightarrow (\overline{\overline{\alpha}})rc$, then there is some type equivalent to r in `deval` c rc. This is used only in the case of `infer` for `case` statements that appears below. Because there are finitely many possible inputs to `deval`, it can be evaluated quickly by table lookup at type inference time. The implementation does this.

```
fun deval c rc =
   let val u = the unique u such that c ::def u ↪ (ᾱ̄)tc
   in
      {r | r ∈ allrefs u and c ::def r ↪ (ᾱ̄)rc}
   end
```

The new cases of the `infer` algorithm are:

```
fun infer VR (c[r̄̄] e') =
      let val k? = infer VR e'
          val t = the unique t such that c ::def t ↪ tc
          val s = allrefs (t̄̄)
      in
        △fn {(r̄̄)rc | r̄̄ ∈ s and for some r ∈ allrefs t we have
                  c ::def r ↪ (ᾱ̄)rc and subtypep k? ([r̄̄/ᾱ̄]r) t}
      end
  | infer VR (e as (case e0 of c1 => e1 | ... | cn => en end:t)) =
      if not rtom(VR) ⊢ e :: t then ns
      else let val r? = infer VR e0
               val u = the unique u such that rtom(VR) ⊢ e0 :: u
           in if r? = ns then ns
              else let val (r̄̄1)rc1 ∧ ... (r̄̄m)rcm = r?
                       fun seq h = (deval ch rc1 × ... × deval ch rcm)
                   in
                      sjoinf t
                         {ifn (infer VR eh, [r̄̄1/ᾱ̄]r1 ∧ ... ∧ [r̄̄m/ᾱ̄]rm) u |
                            h ∈ 1...n and (r1, ..., rm) ∈ (seq h)}
                   end
           end
```

A full proof of this would require replacing two of the cases in each of the proofs of Theorem 2.100 (Infer Returns Some Type) on page 145, Theorem 2.101 (Infer Returns

Principal Type) on page 151, and Theorem 2.102 (Infer Terminates) on page 160. Most of these new cases would be very similar to the cases they replaced, so we shall omit them. The exception is the case of Infer Returns Principal Type for `case` statements, which we give below, after a lemma.

Before we can prove that the algorithm for inferring types for `case` statements returns a principal type, we must show that as the refinement type of the case object becomes stronger, the type inferred for the `case` statement as a whole becomes stronger. To state this formally, we speak in terms of the premises of CASE-TYPE:

Lemma 5.31 (Case Statement Body) *If*

$$(\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_m)rc_m \leq (\overline{\overline{k}}_1)kc_1 \wedge \ldots \wedge (\overline{\overline{k}}_n)kc_n \tag{5.6}$$

and, for all $k_1, \ldots, k_n$ *we have*

$$\begin{array}{l} \textit{for all } h \textit{ in } 1 \ldots n \textit{ we have } c \stackrel{\text{def}}{:} k_h \hookrightarrow (\overline{\overline{\alpha}})kc_h \\ \textit{implies} \\ \text{VR} \vdash e : ([\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge \ldots \wedge [\overline{\overline{k}}_n/\overline{\overline{\alpha}}]k_n) \rightarrow r \end{array} \tag{5.7}$$

and

$$\textit{for all } j \textit{ in } 1 \ldots m \textit{ we have } c \stackrel{\text{def}}{:} r_j \hookrightarrow (\overline{\overline{\alpha}})rc_j, \tag{5.8}$$

then

$$\text{VR} \vdash e : ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m) \rightarrow r.$$

The proof is not particularly interesting, but it is long enough that it was a nuisance to discover, so we include it here.

Proof: Suppose that $\overline{\overline{k}}_h$ has the form

$$\overline{k}_h; \overline{k}'_h; \overline{k}''_h; \overline{k}'''_h$$

and $\overline{\overline{r}}_j$ has the form

$$\overline{r}_j; \overline{r}'_j; \overline{r}''_j; \overline{r}'''_j$$

and $\overline{\overline{\alpha}}$ has the form

$$\overline{\alpha}; \overline{\alpha}'; \overline{\alpha}''; \overline{\alpha}''',$$

and let p abbreviate $(\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_m)rc_m$. By Fact 5.12 (Bound on Argument to i Gives Bound on i) on page 250 and (5.6),

$$\text{for } h \in 1 \ldots n \text{ we have } i(p)(\overline{k}_h; \overline{k}''_h) \preceq (\overline{k}'_h; kc_h). \tag{5.9}$$

Define

$$s(h) = \{j \in 1 \ldots m \mid \overline{k}_h \leq \overline{r}_j \text{ and } \overline{k}''_h \equiv \overline{r}''_j\}. \tag{5.10}$$

Then, by definition of i and (5.9),

$$\text{for } h \in 1 \ldots n \text{ we have } \triangle\{(\overline{r}'_j; rc_j) \mid j \in s(h)\} \preceq (\overline{k}'_h; kc_h)$$

which implies

$$\text{for } h \in 1 \ldots n \text{ we have } \wedge\{\overline{r}'_j \mid j \in s(h)\} \leq \overline{k}'_h \tag{5.11}$$

and

$$\text{for } h \in 1 \ldots n \text{ we have } \overset{\text{def}}{\wedge}\{rc_j \mid j \in s(h)\} \overset{\text{def}}{\leq} kc_h. \tag{5.12}$$

We can use (5.8) and Assumption 2.52 (Constructor Argument Strengthen) on page 67 to get

$$\text{for } h \in 1 \ldots n \text{ and all } j \text{ in } s(h) \text{ we have } c \overset{\text{def}}{:} \wedge\{r_j \mid j \in s(h)\} \hookrightarrow (\overline{\overline{\alpha}})rc_j$$

and then Assumption 2.51 (Constructor And Introduction) on page 67 gives

$$\text{for } h \in 1 \ldots n \text{ we have } c \overset{\text{def}}{:} \wedge\{r_j \mid j \in s(h)\} \hookrightarrow (\overline{\overline{\alpha}})(\overset{\text{def}}{\wedge}\{rc_j \mid j \in s(h)\})$$

and Assumption 2.53 (Constructor Result Weaken) on page 67 with (5.12) then gives

$$\text{for } h \in 1 \ldots n \text{ we have } c \overset{\text{def}}{:} \wedge\{r_j \mid j \in s(h)\} \hookrightarrow (\overline{\overline{\alpha}})kc_h. \tag{5.13}$$

Define k_h to mean $\wedge\{r_j \mid j \in s(h)\}$. Then by (5.7), we have

$$\text{VR} \vdash e : ([\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge \ldots \wedge [\overline{\overline{k}}_n/\overline{\overline{\alpha}}]k_n) \to r. \tag{5.14}$$

The remainder of the proof consists of showing that

$$([\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge \ldots \wedge [\overline{\overline{k}}_n/\overline{\overline{\alpha}}]k_n) \to r \leq ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m) \to r.$$

Once we prove this, we can use WEAKEN-TYPE with (5.14) to get our conclusion.

By Assumption 5.27 (Variance) on page 258, for $h \in 1 \ldots n$ and all $j \in s(h)$ we have $\overline{\overline{\alpha}}$ varies properly in r_j. By definition of $s(h)$, we have

$$\text{for } h \in 1 \ldots n \text{ we have } j \in s(h) \text{ implies } \overline{k}_h \leq \overline{r}_j$$

and

$$\text{for } h \in 1 \ldots n \text{ we have } j \in s(h) \text{ implies } \overline{k}''_h \equiv \overline{r}''_j.$$

By SELF-SUB,

$$\text{for } h \in 1 \ldots n \text{ we have } j \in s(h) \text{ implies } \overline{r}'_j \leq \overline{r}'_j.$$

Thus by the definition of $\leq$ for quadruples we have

$$\text{for } h \in 1 \ldots n \text{ and all } j \in s(h) \text{ we have } \overline{\overline{r}}_j \leq \overline{k}_h; \overline{r}'_j; \overline{k}''_h; \overline{k}'''_h.$$

Then Fact 5.28 (Variant Weakening) on page 258 gives

$$\text{for } h \in 1 \ldots n \text{ and all } j \in s(h) \text{ we have } [\overline{\overline{r}}_j/\overline{\overline{\alpha}}]r_j \leq [\overline{k}_h; \overline{r}'_j; \overline{k}''_h; \overline{k}'''_h/\overline{\overline{\alpha}}]r_j.$$

Then

$$\begin{array}{l}\text{for } h \in 1 \ldots n \text{ we have} \\ \wedge\{[\overline{\overline{r}}_j/\overline{\overline{\alpha}}]r_j \mid j \in s(h)\} \leq \\ \wedge\{[\overline{k}_h; \overline{r}'_j; \overline{k}''_h; \overline{k}'''_h/\overline{\overline{\alpha}}]r_j \mid j \in s(h)\}\end{array} \tag{5.15}$$

because each element of the set on the left hand side is a subtype of the corresponding element of the set on the right hand side.

It is easy to use induction to generalize Assumption 5.29 (Predefined Intersection Distributivity) on page 259 to apply when more than two refinement types are involved in the intersection; thus we have

$$\begin{array}{l}\text{for } h \in 1 \ldots n \text{ we have} \\ \wedge\{[\overline{r}'_j/\overline{\alpha}]r_j \mid j \in s(h)\} \leq \\ \quad [\wedge\{\overline{r}'_j \mid j \in s(h)\}/\overline{\alpha}'](\wedge\{r_j \mid j \in s(h)\})\end{array}$$

Then Fact 5.28 (Variant Weakening) on page 258 used with (5.11) gives

$$\begin{array}{l}\text{for } h \in 1 \ldots n \text{ we have} \\ [\wedge\{\overline{r}'_j \mid j \in s(h)\}/\overline{\alpha}'](\wedge\{r_j \mid j \in s(h)\}) \leq [\overline{k}'_h/\overline{\alpha}](\wedge\{r_j \mid j \in s(h)\}).\end{array}$$

Then TRANS-SUB applied to these gives

$$\text{for } h \in 1 \ldots n \text{ we have } \wedge\{[\overline{r}'_j/\overline{\alpha}']r_j \mid j \in s(h)\} \leq [\overline{k}'_h/\overline{\alpha}'](\wedge\{r_j \mid j \in s(h)\}).$$

By Fact 4.2 (Type Substitution Preserves Subtyping) on page 233, this implies

$$\begin{array}{l}\text{for } h \in 1 \ldots n \text{ we have} \\ [\overline{k}_h; \overline{k}''_h; \overline{k}'''_h/\overline{\alpha}; \overline{\alpha}''; \overline{\alpha}'''](\wedge\{[\overline{r}'_j/\overline{\alpha}']r_j \mid j \in s(h)\}) \leq \\ [\overline{k}_h; \overline{k}''_h; \overline{k}'''_h/\overline{\alpha}; \overline{\alpha}''; \overline{\alpha}'''][\overline{k}'_h/\overline{\alpha}'](\wedge\{r_j \mid j \in s(h)\}),\end{array}$$

and the definitions of substitution and k_h then give

$$\begin{array}{l}\text{for } h \in 1 \ldots n \text{ we have} \\ \wedge\{[\overline{k}_h; \overline{r}'_j; \overline{k}''_h; \overline{k}'''_h/\overline{\alpha}; \overline{\alpha}'; \overline{\alpha}''; \overline{\alpha}''']r_j \mid j \in s(h)\} \leq \\ [\overline{\overline{k}}_h/\overline{\overline{\alpha}}]k_h\end{array}$$

Then TRANS-SUB with (5.15) gives

$$\text{for } h \in 1 \ldots n \text{ we have } \wedge\{[\overline{\overline{r}}_j/\overline{\overline{\alpha}}]r_j \mid j \in s(h)\} \leq [\overline{\overline{k}}_h/\overline{\overline{\alpha}}]k_h.$$

By repeated use of AND-ELIM-L-SUB and AND-ELIM-R-SUB, this implies

$$\text{for } h \in 1 \ldots n \text{ we have } [\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m \leq [\overline{\overline{k}}_h/\overline{\overline{\alpha}}]k_h,$$

and by repeated use of AND-INTRO-SUB, this implies

$$[\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_h \leq [\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge \ldots \wedge [\overline{\overline{k}}_n/\overline{\overline{\alpha}}]k_n.$$

Finally, we can use RCON-SUB to infer

$$([\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge \ldots \wedge [\overline{\overline{k}}_n/\overline{\overline{\alpha}}]k_n) \to r \leq ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_h) \to r$$

and then WEAKEN-TYPE with this and (5.14) gives our conclusion. □

Now we can give the `case` case of Theorem 2.101 (Infer Returns Principal Type) on page 151. We will restate the theorem first.

Theorem 5.32 (Infer Returns Principal Type) *If*

all splits of types in VR *are useless*

and

`infer` VR e *terminates*

and

$$\text{VR} \vdash e : r$$

then

$$(\texttt{infer VR } e) \preceq r$$

Proof: By induction on e. As in Chapter 2 on page 152, we will use

$$\begin{array}{l}\text{For all } r,\\ (\text{VR} \Vdash e : r \text{ implies}\\ \quad (\texttt{infer VR } e) \preceq r)\\ \text{implies}\\ \text{For all } r,\\ (\text{VR} \vdash e : r \text{ implies}\\ \quad (\texttt{infer VR } e) \preceq r)\end{array} \tag{5.16}$$

Case: $e \propto$ `case` e_0 `of` c_1 `=>` e_1 `|` ... `|` c_n `=>` e_n `end:`t Suppose we have an r such that

$$\text{VR} \Vdash e : r. \tag{5.17}$$

The last inference of this must be CASE-TYPE with the premises

$$\text{VR} \vdash e_0 : (\overline{\overline{k}}_1)kc_1 \wedge \ldots \wedge (\overline{\overline{k}}_z)kc_z,$$

$$r \sqsubset t,$$

for all $h \in 1 \ldots n$ and all $k_1, \ldots, k_m$, whenever

for all $q \in 1 \ldots z$ we have $c_h \overset{\text{def}}{:} k_q \hookrightarrow (\overline{\overline{\alpha}})kc_q$

we have

$$\text{VR} \vdash e_h : ([\overline{\overline{k}}_1/\overline{\overline{\alpha}}]k_1 \wedge \ldots \wedge [\overline{\overline{k}}_z/\overline{\overline{\alpha}}]k_z) \to r,$$

and

$$\text{rtom(VR)} \vdash e :: t.$$

Since `infer` VR e terminates, `infer` VR e_0 must terminate. By induction hypothesis,

$$\texttt{infer VR } e_0 \leq (\overline{\overline{k}}_1)kc_1 \wedge \ldots \wedge (\overline{\overline{k}}_z)kc_z.$$

Suppose `infer` VR e_0 has the form $(\overline{\overline{r}}_1)rc_1 \wedge \ldots \wedge (\overline{\overline{r}}_m)rc_m$. We will show that for all h and all $(r_1, \ldots, r_n)$ in `seq` h, we have

$$\texttt{ifn } (\texttt{infer VR } e_h\texttt{, } [\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_h) \leq r.$$

Then trivial properties of `join` will give `infer` VR $e \leq r$, which is our conclusion.

First, choose any h in $1 \ldots n$ and $(r_1, \ldots, r_m)$ in `seq` h. By the definition of `seq`, this implies

$$\text{for } j \in 1 \ldots m \text{ we have } c_h \overset{\text{def}}{:} r_j \hookrightarrow (\overline{\overline{\alpha}})rc_j.$$

Lemma 5.31 (Case Statement Body) on page 265 then gives

$$\text{VR} \vdash e_h : ([\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}])r_n \rightarrow r.$$

Since `infer` VR e terminates, `infer` VR e_h must also terminate. Our induction hypothesis then gives

$$\texttt{infer VR } e_h \leq [\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m \rightarrow r.$$

Hence, by Fact 5.12 (Bound on Argument to i Gives Bound on i) on page 250 we have

$$\texttt{ifn } (\texttt{infer VR } e_h\texttt{, } [\overline{\overline{r}}_1/\overline{\overline{\alpha}}]r_1 \wedge \ldots \wedge [\overline{\overline{r}}_m/\overline{\overline{\alpha}}]r_m) \preceq r.$$

Since this holds for all h and all $(r_1, \ldots, r_m)$ in `seq` h, soundness of `sjoinf` gives `infer` VR $e \leq r$. Summarizing (5.17) to here,

$$\text{VR} \Vdash e : r \text{ implies } \texttt{infer VR } e \preceq r$$

By (5.16), this implies our conclusion.

Case: Otherwise. Omitted. □

5.8 Declaring Polymorphic Type Constructors

Three new issues arise when analyzing `rectype` declarations with polymorphic type constructors: we must determine which type arguments are mixed, positive, negative, and ignored; we must ensure that Assumption 5.29 (Predefined Intersection Distributivity) on page 259 holds; and we must construct default refinement types that expressions written without concern for refinement types can inhabit.

Except for these issues, analyzing `rectype` declarations with type variables is very similar to analyzing the same declarations with all the variables replaced by constants, so the theory from Chapter 3 applies directly.

5.8.1 Separating Mixed, Positive, Negative, and Ignored

We can use a straightforward abstract interpretation of the type declaration to distinguish the different kinds of arguments to type constructors. We separately infer whether a type variable appearing as an argument occurs positively and whether it occurs negatively in the definition of the type; if neither is true, the argument is ignored, and if both are true, the argument is mixed. For example, given the datatypeq

$$\texttt{datatype } (\alpha, \beta)\ mix \texttt{ = Mix of } \alpha \rightarrow (\alpha * \beta)$$

we immediately determine that α is mixed and β is positive. When several mutually recursive type constructors are declared simultaneously, we may have to iterate to determine the best classification. For example, given the declaration

$$\begin{array}{l}\texttt{datatype } (\alpha, \beta)\ m1 \texttt{ =} \\ \qquad\qquad \texttt{A of } (\alpha, \beta)\ m2 \texttt{ | B of } \alpha * \alpha \\ \qquad \texttt{and } (\gamma, \delta)\ m2 \texttt{ =} \\ \qquad\qquad \texttt{C of } (\gamma, \delta)\ m1 \texttt{ | D of } \delta \rightarrow \delta\end{array}$$

we will have to use at least two iterations to determine that α and γ are positive and β and δ are mixed. Implementing this is straightforward.

5.8.2 Enforcing Predefined Intersection Distributivity

The best way to enforce Assumption 5.29 (Predefined Intersection Distributivity) on page 259 is unclear. The following theory says it is possible to effectively discover declarations for which this assumption is not true; we could simply reject them, but it would be better to silently repair declarations to cause the rule to be true. It is not obvious how to repair the declarations.

The following fact has an immediate corollary which leads to an algorithm that recognizes declarations for which the assumption is not true. The main idea is that we can determine whether the assumption is true for all vectors of refinement types we could substitute by checking it in one special case. The special case uses a vector $\overline{tc}$ of monomorphic ML type constructors and two vectors $\overline{a}$ and $\overline{b}$ of monomorphic refinement type constructors, where all three vectors have the same length and the type constructors in all three of these vectors are distinct from each other and from any other type constructors mentioned in the lemma, and for all $i \in 1 \ldots \text{length}(\overline{tc})$, the only refinements of $\overline{tc}\,\{i\}$ are $\overline{a}\,\{i\}$, $\overline{b}\,\{i\}$, and $\overline{a}\,\{i\} \stackrel{\text{def}}{\wedge} \overline{b}\,\{i\}$. Similarly, we use the vectors $\overline{\alpha}$, $\overline{\alpha}'$, and $\overline{\alpha}''$ where all three vectors have the same length as $\overline{tc}$ and no type variable appears more than once in all three vectors. Then we have:

Fact 5.33 (Predefined Intersection Distributivity Technical) *Suppose $\overline{a}$ and $\overline{b}$ are as described above, and that $\overline{k}$ and $\overline{k}'$ are vectors of refinement types where $\overline{k} \wedge \overline{k}'$ refines some $\overline{t}$. Let*

$$s = [\overline{a}/\overline{\alpha}, \overline{b}/\overline{\alpha}', (\overline{a} \wedge \overline{b})/\overline{\alpha}'']$$

and

$$s' = [\overline{k}/\overline{\alpha}, \overline{k}'/\overline{\alpha}', (\overline{k} \wedge \overline{k}')/\overline{\alpha}''].$$

Then, for any refinement types k and k' in which none of the $a\{i\}$'s or $b\{i\}$'s appear, if

$$s(k) \leq s(k')$$

then

$$s'(k) \leq s'(k').$$

The proof of this is a straightforward induction on the derivation of $s(k) \leq s(k')$. Then we have the following corollary:

Corollary 5.34 (Predefined Intersection Distributivity Decidable) *Suppose $\overline{a}$ and $\overline{b}$ are as described above, and that $\overline{k}$ and $\overline{k}'$ are vectors of refinement types where $\overline{k} \wedge \overline{k}'$ refines some $\overline{t}$. Then, for any refinement types k and k' in which none of the $a\{i\}$'s or $b\{i\}$'s appear, if*

$$[\overline{a}/\overline{\alpha}]r \wedge [\overline{b}/\overline{\alpha}]r' \leq [\overline{a} \wedge \overline{b}/\overline{\alpha}](r \wedge r')$$

then

$$[\overline{k}/\overline{\alpha}]r \wedge [\overline{k}'/\overline{\alpha}]r' \leq [\overline{k} \wedge \overline{k}'/\overline{\alpha}](r \wedge r').$$

Proof: Use the previous fact, with $k = r \wedge [\overline{\alpha}'/\overline{\alpha}]r'$ and $k' = [\overline{\alpha}''/\overline{\alpha}](r \wedge r')$. □

At this point, an admittedly slow algorithm for checking the assumption is clear: each time we analyze a `rectype` declaration, for each constructor c where

$$c \stackrel{\text{def}}{::} t \hookrightarrow (\overline{\alpha}_1; \overline{\alpha}_2; \overline{\alpha}_3; \overline{\alpha}_4)tc,$$

temporarily introduce new $\overline{tc}$, $\overline{a}$, and $\overline{b}$ as described in the corollary above. Enumerate all r, r', rc, and rc' such that $c \stackrel{\text{def}}{::} r \hookrightarrow (\overline{\overline{\alpha}})rc$ and $c \stackrel{\text{def}}{::} r' \hookrightarrow (\overline{\overline{\alpha}})rc'$; in each case, verify that $[\overline{a}/\overline{\alpha}_2]r \wedge [\overline{b}/\overline{\alpha}_2]r' \leq [\overline{a} \wedge \overline{b}/\overline{\alpha}_2]$.

There may be faster algorithms for doing this test, and there may be ways to repair `rectype` declarations that fail this test without violating the intuitive expectations of the programmer. All this is future work.

5.8.3 Default Refinement Types

As the example in Subsection 2.7.2 on page 74 shows, if any ML type has more than one refinement, there will be programs with an ML type that have no refinement type. Once we permit mixed type arguments, we get into an even stranger situation: there will be ML types with no maximal refinement. For example, assuming the usual refinements of the booleans and the declared ML type m on page pagerefexample:mixed, the refinements of $bool\ m$ are $tt\ \top_m$, $ff\ \top_m$, $\bot_{bool}\ \top_m$, $\top_{bool}\ \top_m$, and intersections of these. There is no refinement of $bool\ m$ that is greater than both $\top_{bool}\ \top_m$ and $tt\ \top_m$.

Since there is no refinement type that includes the others, the terminology "catch-all type" that we used in previous chapters is not appropriate. Instead, the purpose of the default type is to provide a refinement type that terms with an ML type can inhabit whenever the strangeness from Subsection 2.7.2 does not happen. With this understanding, constructing the default type of a given ML datatype is still straightforward: define the default refinement type to be any constructor that constructs the ML datatype, applied to the default type refining the argument ML type of the constructor. For example, given the datatype

$$\texttt{datatype}\ (;;\alpha;)\ mix\ \texttt{=}\ \texttt{Mix of}\ \alpha \rightarrow (\alpha * bool)$$

the default refinement type is defined as

$$\texttt{rectype}\ (;;\alpha;)\ \top\_mix\ \texttt{=}\ \texttt{Mix}\ (\alpha \rightarrow (\alpha * \top_{bool}))$$

Chapter 6

Declaring Refinement Types for Expressions

In this chapter we add explicit refinement type declarations to the language of expressions; for example, the expression

$$\texttt{fn x:}\mathit{bool}\ \texttt{=>}\ (\texttt{x[]} \lhd tt)$$

will have the type $tt \to tt$ but not the type $\mathit{ff} \to \mathit{ff}$. Adding this feature is surprisingly simple.

The $\lhd$ operator is coercive, in the sense that the best refinement type of a expression of the form $e \lhd r$ will be r, if it has any type at all. We can also imagine a non-coercive version, which we shall call $\lhd'$. The best type of $e \lhd' r$ would be the best type of e, if that type is less than r; otherwise the expression has no type.

Both operators are simple, but $\lhd$ is more elegant because we can use $\lhd$ to implement $\lhd'$, but not vice versa. To use $\lhd$ to implement $\lhd'$, regard the expression $e \lhd' r$ as an abbreviation for $(\texttt{fn z:}t\ \texttt{=>}\ ((\texttt{fn x:}t\ \texttt{=>}\ \texttt{fn y:}t\ \texttt{=>}\ \texttt{x[]})\ \texttt{z[]}\ (\texttt{z[]} \lhd r)))\ e$, where t is a valid ML type for e.

Type inference for $\lhd$ is simple. First we add the syntax to the language; we will still have a use for expressions without any $\lhd$ operators, so we will keep the metavariable e with the meaning it was given on page 242 and use the metavariable d to stand for expressions that may have $\lhd$ operators. Thus the grammar for d is:

$$\begin{aligned} d ::= {} & d \lhd r \mid \\ & x[\overline{t}] \mid \texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ d \mid d\ \ d \mid c[\overline{\overline{t}}]\ \ d \mid \\ & \texttt{case}\ d\ \texttt{of}\ c\ \texttt{=>}\ d\ \texttt{|}\ \ldots\ \texttt{|}\ c\ \texttt{=>}\ d\ \texttt{end:}t \mid \\ & (d, \ldots, d) \mid () \mid \texttt{elt\_}m\texttt{\_}n\ \ d \mid \\ & \texttt{fix}\ f\texttt{:}t\ \texttt{=>}\ \texttt{fn}\ x\texttt{:}t\ \texttt{=>}\ d \mid \\ & \texttt{let}\ x\ \texttt{=}\ \Lambda(\overline{\alpha}).d\ \ \texttt{in}\ \ d\ \ \texttt{end} \end{aligned}$$

Our language of types is unchanged, so results such as Fact 5.21 (Finite Refinements) on page 252 continue to hold. Type inference for expressions with refinement type declarations is the same as type inference for expressions without refinement type declarations, except we add this rule:

$$\text{DECL-TYPE: } \frac{\text{VR} \vdash d : r}{\text{VR} \vdash (d \lhd r) : r}$$

Strictly speaking, we also need to take the rules in effect for e (see page 254) and assert that they still hold, except all e's in those rules should be changed to d's.

Instead of using coercive declarations to implement non-coercive declarations, we could treat non-coercive declarations directly by adding this rule:

$$\text{DECL-TYPE': } \frac{\text{VR} \vdash d : r \qquad r \leq k}{\text{VR} \vdash (d \lhd' k) : r}$$

The presence or absence of this rule has little impact on the reasoning below.

With these declarations there comes a new phenomenon: expressions can now have free refinement type variables. Attempts to directly define a notion of evaluation on expressions with declarations lead to pointless questions about how to instantiate free refinement type variables while evaluating `let` statements. To avoid these questions, we simply erase the refinement type declarations before evaluating:

Definition 6.1 (Erase) *We use the notation* erase(d) *to mean* d *with all of the refinement type declarations erased.*

Our soundness result therefore reads as follows:

Theorem 6.2 (Refinement Type Soundness) *If* erase$(d) \Rightarrow v$ *and* $\cdot \vdash d : r$*, then* $\cdot \vdash v : r$.

Proof: We can use induction to prove that $\cdot \vdash d : r$ implies $\cdot \vdash$ erase$(d) : r$. Thus $\cdot \vdash$ erase$(d) : r$, and we can apply Theorem 4.7 (Refinement Type Soundness) on page 237 to get our conclusion. □

The algorithm for inferring refinement types for expressions with refinement type declarations is also simple. We add the following case:

```
fun infer VR d ◁ r =
   if rtom(VR) ⊢ erase(d) :: t then
      let t = the unique t such that rtom(VR) ⊢ erase(d) :: t
      in
         if subtypep (infer VR d) r t then r else ns
      end
   else ns
```

The soundness proof for this is straightforward and we omit it.

Chapter 7

Implementation

An implementation of refinement type inference has been written in Standard ML. It corresponds well with the theory developed in the previous chapters, and it runs reasonably quickly. This chapter discusses the technical issues that had to be resolved to create this implementation; this chapter is not meant to be complete instructions for using the implementation.

Since the language of the implementation resembles the object language, there is potential for confusion between the object and the implementation languages. Worse, examining types of expressions in the implementation language is useful when trying to understand the implementation, so we must add yet another kind of type to the discussion. We call types in the implementation language "SML types", to distinguish them from the "ML types" in the object language described in the previous chapters.

The syntax for the expressions recognized by the implementation is similar to the grammar appearing on page 274, except we implement ML type inference so explicit ML types need not appear in terms. The grammar does not closely resemble true SML. A simple interaction with the implementation is below. In the example, the refinement type declaration operator "$\lhd$" is written as "`<:`" and the operator "$\rightarrow$" is written as "`->`". Input typed by the user is preceded by `>-` or `>=`.

```
>- datatype bool = true of unit | false of unit
>= rectype tt = true (unit) and ff = false (unit);
>- true;
it: (unit -> bool) ::
(unit -> tt)
>- true <: ff;
Failed to unify (unit -> bool) and bool
     while ML type checking true <: ff.
ML type check failed.
>- (true ()) <: ff;
it: bool ::
<: invalid for term (true[] ()).
It actually had the type:
tt
You tried to coerce it to the type ff.
>- (true ()) <: tt;
it: bool ::
tt
>-
```

As is the case in the theory described in previous chapters, all value constructors take exactly one argument. Notice that at no point do we calculate a value; this implementation of refinement type inference does not implement any kind of evaluation.

The implementation has many boolean flags that the user can manipulate to turn on and off various performance optimizations in the type checker. The flags are all false by default; the flags are defined in such a way that the default is usually best. A given flag f can be set with the top level command "`setflag` f;" or cleared with the top level command "`clearflag` f;". A list of all flags with a description and the present value of each is printed whenever the flag argument to `setflag` or `clearflag` is invalid.

Most of the optimizations discussed below can be turned off by setting some flag. We justify most optimizations to type inference by citing how turning off the optimization makes some example run more slowly. All run times in this chapter were measured on a SPARCstation iPX.

7.1 Representations

This section discusses how the various mathematical objects discussed in previous chapters are represented in the implementation.

7.1.1 Type Constructors

The *Definition of Standard ML* [MTH90] makes a distinction between a type name and a type identifier. Type identifiers are simply the strings appearing in the program text that refer to various types; for example, if we have a declaration of the form `type t = ...` that is shadowed by a later declaration of the form `datatype t = ...`, both of the types have the same identifier `t`. In contrast, type names are unique for each type; since the two types with identifier `t` are different, they will have different type names.

In the implementation refinement types we make even more distinctions. First we have refinement type identifiers (`refconid`'s) and ML type identifiers (`mlconid`'s), which are both implemented as `strings`. Then we have refinement type names (`refconname`'s) and ML type names (`mlconname`'s), which are unique identifiers. These are equality types, so they can easily be used as keys for tables. Finally, we have ML type constructors (`mlconstructor`'s), which have an `mlconname` and other information describing all the refinements of that ML type name.

Refinement type identifiers and names are represented as follows:

```
type refconid = string
type refconname = {refconid : refconid, uniqueid : int, index : int}
```

The `uniqueid` field of `refconname`'s is used to distinguish different refinement type constructors with the same name. If an ML type constructor tc is refined by the refinement type constructors $rc_1, \ldots, rc_n$, then the `index` field of the representations of these refinement type constructors will be distinct integers in the range $0, \ldots, n-1$, in some order. This allows us to implement functions mapping a refinement of tc to some other value as a simple array reference.

ML type identifiers and names are represented as follows:

```
type mlconid = string
type low_mlconname = mlconid * int
datatype mlconname =
    Tuple of int
  | Arrow
  | Custom of low_mlconname
```

This is all very straightforward: `Tuple` n stands for $ttuple_n$, `Arrow` stands for $tarrow$, and `Custom` $(s,\ i)$ stands for the user-defined ML type constructor with the name s. The integer i has the same role as the `uniqueid` field of refinement type constructor names.

Note that `mlconname`'s distinguish separate cases for arrow and tuple types, but `refconname`'s do not. This does not create ambiguity because whenever the implementation uses a refinement type constructor, it always has on hand the ML type constructor that this refinement type constructor refines. Thus if a refinement type constructor refines

an ML type constructor with the name `Arrow`, it must be the representation of $rarrow$. In this case we put arbitrary values in the `refconid`, `uniqueid`, and `index` fields. We do the same for tuple refinement type constructors.

The ML type constructor itself is a record containing the ML type constructor name, along with other information describing its refinements. The meanings of the fields are discussed below.

```
datatype mlconstructor =
    MLConstr of
    {name : mlconname,
     unique_refinements : refconname list,
     nonunique_refinements : (refconname, refconname) S.substitution,
     bottom : refconname,
     bottom_emptyp : bool,
     top : refconname,
     tor : (refconname * refconname) -> refconname,
     tand : (refconname * refconname) -> refconname,
     tleq : (refconname * refconname) -> bool,
     negargpos : int list,
     posargpos : int list}
```

The `unique_refinements` and `nonunique_refinements` fields are used to keep redundant refinements of an ML type from slowing refinement type inference. For example, if this declaration is given to refinement type inference:

$$\begin{array}{l}\texttt{datatype } bool \texttt{ = true of } tunit \texttt{ | false of } tunit\\ \texttt{rectype } tt \texttt{ = true } (tunit)\\ \qquad \texttt{and } ff \texttt{ = false } (tunit)\\ \qquad \texttt{and } \bot_{bool} \texttt{ = bottom } bool;\end{array}$$

then the refinement types $tt \overset{\text{def}}{\wedge} ff$ and $\bot_{bool}$ will be equivalent. The `unique_refinements` field has a list of the refinements we will use (which excludes $tt \overset{\text{def}}{\wedge} ff$), and `nonunique_refinements` has a substitution mapping each refinement we will not use into the corresponding one we will use (in this case, $tt \overset{\text{def}}{\wedge} ff$ is mapped to $\bot_{bool}$).

Skipping forward, the `tor`, `tand`, and `tleq` fields contain functions that can join, intersect, and compare the refinements of this ML type constructor. The functions `tor` and `tand` only have elements of `unique_refinements` as their range, to make it possible to pay as little attention as possible to the redundant refinement type constructors.

The fields `bottom` and `top` have the least and greatest refinement of this ML type constructor, respectively. These fields are redundant; we could compute them by using the functions stored in the `tand` and `tor` fields to combine the types listed in `nonunique_refinements`.

The `bottom_emptyp` field is used to evaluate the $\vdash r$ empty judgement described in Chapter 3. If this flag is true, we assume that the refinement in the `bottom` field is empty, otherwise we assume it is not. We assume that all nonredundant refinements other than the one in the `bottom` field are not empty, since otherwise they would be equivalent to the refinement in the `bottom` field, and therefore they would be redundant.

All type arguments in the implementation are either positive or negative. Mixed arguments are outlawed and ignored arguments are treated as positive. Syntactically, each ML type constructor has one linear list of arguments, as they do in SML; but internally, we treat the negative type arguments very differently from the positive ones, so we keep them segregated into separate lists. The `posargpos` and `negargpos` fields say how to do the segregation. If we sequentially assign numbers (starting with zero) to the syntactic type arguments, then `posargpos` is a list of the numbers for positive type arguments and `negargpos` is a list of the numbers for negative type arguments. For example, given the declaration

$$\texttt{datatype } (\alpha, \beta, \gamma) \ d \texttt{ = D of } \beta * (\alpha \rightarrow \gamma)$$

argument number 0 (α) is negative and arguments 1 (β) and 2 (γ) are positive, so the `negargpos` field of d will be `[0]` and the `posargpos` field will be `[1, 2]`.

7.1.2 ML types and type schemes

We represent ML types with the datatype

```
datatype mlty =
    MLCon of {neg : mlty list,
              pos : mlty list,
              con : mlconstructor}
  | MLTyvar of V.tyvar
```

where `V.tyvar` is a representation of type variables. This is a direct encoding of the grammar for ML types given on page 242, except we have already segregated the negative and positive type arguments; from the `negargpos` and `posargpos` fields of the constructor, it is obvious how to merge the `neg` and `pos` fields to get the type arguments in the order the user expects.

The encoding of ML type schemes is straightforward as well:

```
datatype mlscheme =
    MLScheme of (V.tyvar list * mlty)
```

These encodings of ML types are straightforward enough that we will ignore them in this chapter, and use the same notation for ML types in this chapter that we have used in previous chapters.

7.1.3 Refinement Types

The implementation has different representations for the refinement types appearing in explicit declarations (such as tt in $(\texttt{true}\ ())\ \lhd\ tt$) and refinement types that are inferred for an expression by the implementation. After we describe both representations, we will explain below how the special representation for refinement types in explicit declarations allows quick checking of the assertions in expressions containing $\lhd$.

The representation for refinement types appearing in explicit declarations is simple, and similar to the representation of ML types:

```
datatype syntp =
    SAnd of syntp list
  | SCon of {pos : syntp list,
             neg : syntp list, refcon : refconname}
  | SVar
```

Since we usually know which ML type a refinement type refines, we only have one representation `SVar` for all type variables appearing in explicitly-declared refinement types.

We represent inferred refinement types with a function that computes the interpretation i of the refinement type as in Definition 5.8 on page 248, along with a few other fields that make some optimizations possible. The representation of refinement types is:

```
datatype tp =
    RefCon of (teqopt * bool * mlconstructor *
               (tp list -> (tp list * refconname)))
  | Reftyvar
```

`Reftyvar` is analogous to `SVar`; it stands for a type variable, but it does not bother to say which one, because there is generally an ML type on hand that makes that clear. If the refinement type is not a type variable, then the constructor is `RefCon` with a tuple of four components as its argument.

Skipping ahead, the fourth component of the tuple is the interpretation, represented as a function in the obvious way. We only have one argument to the function because we outlaw mixed type variables.

The first component of the tuple has the type `teqopt` which we have not yet discussed. This type is used for memoizing refinement type equality, and it is discussed with memoization in Subsection 7.2.4 below. In practice, the implementation uses a utility procedure called `eRefCon` that inserts the `teqopt`; `eRefCon` takes as argument a tuple with the last three components of the argument to `RefCon`, it constructs and inserts an appropriate `teqopt`, and it calls `RefCon` and returns the resulting `tp`.

The implementation of refinement types uses references when it finds a type for a fixed point. As the values stored in these references change, the behavior of the functional

component of some refinement types can change; we say these refinement types are not constant. It is important not to memoize these refinement types, because the information stored in the memo table may not be accurate by the time it is used. To ensure this, we use the second component of the tuple in each refinement type created by `RefCon` to record whether the refinement type is constant. For a more complete discussion, see Subsection 7.2.2.

The third component of the tuple is the ML type constructor. This is redundant and could be eliminated; it is presently used so we can recognize arrow and tuple types when printing refinement types during debugging, and so we can form intersections and joins of refinement types without having to pass around the ML type that they refine.

Given the functional component of a `tp` and a `syntp`, one can efficiently determine whether the `tp` is a subtype of the `syntp`. If the `syntp` is an intersection, then the `tp` is a subtype of the `syntp` if and only if it is a subtype of all of the components of the intersection. If the `syntp` is `SVar`, then ML type inference should have ensured that the `tp` is `Reftyvar`, so the `tp` is a subtype of the `syntp`. Lastly, if the `syntp` is a `SCon`, then we use the definition of i to convert the negative arguments of the `syntp` into a `tp`, we pass those negative arguments to the functional component of the `tp`, and we recursively compare the result of the function call to the positive arguments of the `syntp`.

7.2 Refinement Type Inference

Refinement type inference is similar to the type inference algorithm described at the ends of Chapters 2, 4, and 5. The main change is the lazy representation of refinement types; this immediately leads to the needs for memoization, pending analysis for fixed points, and an interesting instantiation algorithm. Lazy representations of types also appear in [HM94]. Once these issues are understood, there is little to be gained by writing out the entire algorithm; instead, we will only deal with interesting cases of it below.

7.2.1 Laziness

Often a function will only be used at a few of the types for which it is defined. This tendency is especially strong for higher-order functions, since functional ML types can have so many distinct refinements. For example, assuming the usual `rectype` declaration for the booleans is in effect, the refinement type given to `double` by the declaration

$$\texttt{val double = fn f => fn x => f (f (x:}bool\texttt{));}$$

is an intersection of 112 components. By representing refinement types as functions that can compute the relevant components of the intersection on demand, we can usually avoid computing all 112 components and storing them in memory.

The case of type inference that reflects this principle most clearly deals with abstractions. Using the notation in the original definition of the type inference algorithm in Figure 2.7 on page 142, this is the modified algorithm:

```
| infer VR (fn x:t => e') =
   if there is a u such that rtom(VR)[x := t] ⊢ e' :: u
   then
      let val u = the unique u such that rtom(VR)[x := t] ⊢ e' :: u
          fun do_one r =
             sjoinf u {infer (VR[x := r']) e' | r' ∈ split r}
       in
          RefCon (..., ..., tarrow, fn [x] => ([do_one x], rarrow))
       end
   else ns
```

where we have omitted the first two components of the argument of `RefCon`. In the actual implementation, the ML type u of the entire abstraction is stored in the abstract syntax of the abstraction, so refinement type inference does not have to invoke ML type inference.

7.2.2 Fixed Points

The method for finding least fixed points in the `fix` case of the algorithm in Figures 2.7 and 2.8 on pages 142 and 143 is an instance of a general technique: start with the least possible value, and repeatedly apply the function we want the fixed point of until the result stops changing. This technique does not work well with lazy representations of refinement types because comparing the results of one iteration to the results of the next causes us to evaluate both results completely.

Instead, we use a technique called *pending analysis*. This technique allows one to evaluate the abstract interpretation of a fixed point at any given point; this evaluation examines a minimal number of other points. It is easiest to explain this with an example; for a more formal description, see any of [Jag89, Dix88, You89]. The tables of pending values resemble the minimal function graphs of [JM86].

Suppose we have the declarations

```
datatype α list = cons of α * α list | nil of tunit
rectype α ev = nil (tunit) | cons (α * α od)
    and α od = cons (α * α ev)
    and α em = nil (tunit)
    and α nem = cons (α * α ⊤list)
    and α ⊥list = bottom (α list);
datatype bool = true of tunit | false of tunit
rectype tt = true (tunit)
    and ff = false (tunit)
    and ⊥bool = bottom bool;
```

and (using the concise syntax) the function definitions

```
fun not (true ()) = false ()
  | not (false ()) = true ()
fun boolmap (f:bool -> bool) ((nil ()):bool list) = nil ()
  | boolmap f (cons (hd, tl)) = cons (f hd, boolmap f tl)
```

or, in the formal syntax, the function definitions

```
val not = fn x:bool =>
    case x of true => false | false => true end:bool;
val boolmap =
    fix boolmap:(bool → bool) → bool list → bool list =>
       fn f:bool → bool => fn l:bool list =>
          case l of
             nil => fn _:tunit => nil ()
           | cons => fn p:bool * bool list =>
                     cons (f (elt_1_2 p), boolmap f (elt_2_2 p))
          end:bool list;
```

and suppose we want to find the principal type for

```
boolmap not (cons (true (), nil ())).
```

We start with an abstract interpretation using a strategy very similar to the strategy for actually evaluating the expression. This becomes interesting when we must take steps to ensure that abstract interpretation terminates even in the presence of recursion. The type of `boolmap` has the form

```
RefCon (teqopt1, true, tarrow, boolmapfn'),
```

where `tarrow` is an ML constructor representing arrow types and `boolmapfn'` is some function. Similarly, the type of `not` is some unimportant structure wrapped around a function

we shall call `notfn'`. From the declaration of `RefCon` on page 280, we know that both `boolmapfn'` and `notfn'` have the SML type `tp list -> (tp list * refconname)`. Because the ML type constructor named `Arrow` has one negative argument, the list of `tp`'s passed to `boolmapfn'` and `notfn'` will always have exactly one element. Because `Arrow` has one positive argument, the list of `tp`'s returned from `boolmapfn'` and `notfn'` will also always have exactly one element. Since `Arrow` is refined by only one refinement type constructor, the `refconname` returned from both `boolmapfn` and `notfn` will always be that refinement type constructor. Thus we can represent all the information in `boolmapfn'` and `notfn'` using functions with the SML type `tp -> tp`; we will call these functions `boolmapfn` and `notfn`.

The interior structure of the type of `cons (true (), nil ())` is not relevant for this example, so we will write that type as a mathematical refinement type: $tt\ od$.

We take the behavior of `notfn` as given, and our goal in this example is to describe the behavior of `boolmapfn` when it is passed the arguments `notfn` and $tt\ od$.

If we had no concerns about termination of type inference, we could simply make the abstract interpretation of `boolmap` recur at the same point in the code where `boolmap` itself recurs. We would start with `f` having the type `notfn` and `l` having the type $tt\ od$. (Throughout this scenario, `f` will have the type `notfn`, so we will not mention it again.) We can summarize this situation with the notation

$$\texttt{boolmapfn notfn}\ (tt\ \ od) = ?$$

We can construct an odd length list starting with either an empty list or an nonempty, even length list, so the abstract interpretation would make two recursive calls to the body of `boolmap`: one where `l` has the type $tt\ em$ (this returns immediately with the result $\bot_{bool}\ \ em$) and one where `l` has the type $tt\ (ev \overset{\text{def}}{\wedge} nem)$. We can summarize the current situation with the table

$$\begin{array}{ll}
\texttt{boolmapfn notfn}\ (tt\ \ od) & = ? \\
\texttt{boolmapfn notfn}\ (tt\ \ em) & = \bot_{bool}\ \ em \\
\texttt{boolmapfn notfn}\ (tt\ \ (ev \overset{\text{def}}{\wedge} nem)) & = ?
\end{array}$$

Continuing, the call with argument $tt\ (ev \overset{\text{def}}{\wedge} nem)$ gives rise to a recursive call where `l` has the type $tt\ od$. This is the argument we started with, so if we continue in the fashion we have up to this point, we will have an infinite loop.

The solution to this problem is the essence of pending analysis. Instead of continuing with the recursion, we behave as though the inner recursive call to $\texttt{boolmapfn notfn}\ (tt\ \ od)$ simply returns the least type we have observed so far for the expression $\texttt{boolmapfn notfn}\ (tt\ \ od)$. Since our table lists "?" as the entry corresponding to $\texttt{boolmapfn notfn}\ (tt\ \ od)$, we have not yet observed any types returned from this expression, so we return the least available type for the expression, which is $\bot_{bool}\ \ \bot_{list}$. Under this assumption, the value returned when `l` is $tt\ (ev \overset{\text{def}}{\wedge} nem)$ is $ff\ \ \bot_{list}$. This type is

less than the true type when `l` is $tt\ (ev \overset{\text{def}}{\wedge} nem)$; we will revise it later. This table describes the current situation:

$$\begin{aligned}
&\texttt{boolmapfn notfn}\ (tt\ \ od) &&= ?\\
&\texttt{boolmapfn notfn}\ (tt\ \ em) &&= \bot_{bool}\ \ em\\
&\texttt{boolmapfn notfn}\ (tt\ \ (ev \overset{\text{def}}{\wedge} nem)) &&= ff\ \ \bot_{list}
\end{aligned}$$

Now we can finish this part of the abstract interpretation; we take the join of $\bot_{bool}\ \ em$ and $ff\ \ \bot_{list}$ and apply `cons`, yielding $ff\ \ od$ and the following table:

$$\begin{aligned}
&\texttt{boolmapfn notfn}\ (tt\ \ od) &&= ff\ \ od\\
&\texttt{boolmapfn notfn}\ (tt\ \ em) &&= \bot_{bool}\ \ em\\
&\texttt{boolmapfn notfn}\ (tt\ \ (ev \overset{\text{def}}{\wedge} nem)) &&= ff\ \ \bot_{list}
\end{aligned}$$

Call this table "Generation 1". Although we have the correct answer to the problem we are interested in, this table is peculiar because the solutions to the subproblems listed on the second and third lines are too small. We were lucky this time; in general, at this point in the computation, the proposed solution to the top-level problem can be too small.

This happened because we knew too little when we computed some of the subproblems. A natural approach is to repeat the computation, but whenever a subproblem that would otherwise cause a loop arises, we use the value from Generation 1 instead of the least type available. Doing this results in the correct result for the top-level problem again, and also a correct table:

$$\begin{aligned}
&\texttt{boolmapfn notfn}\ (tt\ \ od) &&= ff\ \ od\\
&\texttt{boolmapfn notfn}\ (tt\ \ em) &&= ff\ \ em\\
&\texttt{boolmapfn notfn}\ (tt\ \ (ev \overset{\text{def}}{\wedge} nem)) &&= ff\ \ (ev \overset{\text{def}}{\wedge} nem)
\end{aligned}$$

Call this "Generation 2". We only know this is a correct table because we have foreknowledge of the correct result; the only way the implementation can determine that this table is correct is by using it to calculate a third generation, and seeing that Generations 2 and 3 are identical.

It is plausible, but not at all obvious, that this procedure gives correct results. For proofs, see [Dix88].

The implementation organizes the table representing these generations as an association list. Searching the association list can be expensive, in general, because comparing types can be expensive. Therefore each entry in the table is a reference that can be updated in place; this avoids the usual accumulation of useless entries in an association list as new entries are added to the beginning.

Unfortunately, this also means that some refinement types have functions embedded in them that make non-trivial use of references. In general, if a subexpression has a free variable that is bound by a surrounding `fix` operator, its type will contain a function that

may change as we search for the fixed point. Putting this type in a memo table would cause problems, because the behavior of the type may change with time. To ensure that these types are never placed in a memo table, each refinement type other than `Reftyvar` has a boolean field that is set to `true` if it definitely uses no references (that is, it is *constant*), and `false` if it may use nontrivial references of this kind. This is the second component of the tuple argument to `RefCon` on page 280.

7.2.3 Optimizing Equality

The above technique requires us to look up types in a table and to determine whether one table is identical to another. Both of these problems rely heavily on determining when types are equal to each other. There are several steps that can be taken to make this efficient.

The straightforward implementation of type equality (based on a generalization of the `subtypep` function from page 119 to operate on refinement type constructors with negative type arguments) is fairly fast for types without any negative arguments because in that case `allrefs` is never used to enumerate the refinements of an ML type. However, whenever we compare refinements of an ML type with non-trivial negative type arguments such as $(bool \to bool) \to bool$, we will have to enumerate all of the refinements of the negative type arguments; in this example, we have exactly one negative type argument $bool \to bool$.

To have as few of these expensive enumerations as possible, we memoize type equality. Whenever a refinement type is not `Reftyvar`, it contains a tuple where the first component has the type `teqopt` which is used for this purpose. The definition of `teqopt` is:

```
datatype teqopt = TeqOpt of {sameas: UF.set,
                             differentfrom: UF.set list ref}
```

This definition is a `datatype` with only one constructor rather than a type abbreviation because SML does not allow type abbreviations in signatures. The type `UF.set` represents equivalence classes; `UF` is a name for a structure with this signature:

```
signature UNIONFIND =
    sig
        type set
        val newset : unit -> set
        val union : set -> set -> unit
        val sameset : set -> set -> bool
    end
```

Think of a `set` here as a name for something. The function `newset` creates a new name, `union` declares that two names really stand for the same thing, and `sameset` reports whether all of the `union`'s done so far imply that two given names stand for the same thing. As the name of the signature implies, this is the classic Union-Find problem, discussed in [AHU74,

page 124]. The names `set` and `union` are part of the standard nomenclature used with that problem. The implementations of these operations run in almost constant time.

With this understanding of `UF.set`, it should be clear how `teqopt`'s are used. Using `sameset` on two `sameas` fields will return `true` if we have already observed that the two refinement types are equal. Using `sameset` to search the `differentfrom` fields will tell us if we have already determined that two refinement types are different. If we have to compare two refinement types, and the `teqopt` fields do not make it clear that they are either equal or unequal, then we do the comparison using whatever expensive enumerations are necessary, and then we update the `teqopt` fields as necessary to record the result of the comparison. As an exception, we do not try to memoize type equality for refinement types that are not constant.

This strategy works well in practice. The most expensive aspect is searching the `differentfrom` fields. We can improve this even further by only memoizing type equality when expensive iterations are involved (that is, when the type constructor has negative type arguments). Since programs often have many refinements of simple types such as tuples, booleans, and lists, and few refinements of higher-order types, this usually helps. In the `boolmap` example above, memoizing type equality would avoid all comparisons of `notfn` with itself when we are searching the generation tables, but no use of the `teqopt` field would be made when we compare the refinements of `bool list`.

This optimization ought to make a difference when evaluating a fixed point requires comparing function objects with large types. This optimization can be turned on and off by setting the `dont_teq_unionfind` flag, and experiments with this flag show that this optimization rarely makes a difference. Memoizing functional refinement types, as discussed below, makes the amount of work saved by this optimization trivial when function types are fairly small.

7.2.4 Memoizing Refinement Types

When analyzing typical programs, the type inference algorithm described in previous chapters often finds the interpretation of a type and then evaluates that interpretation many times at the same point. Since we represent types by their interpretations, we can hope to save time by memoizing these interpretations. This means that after the first time we evaluate the function at a given point, if an occasion to evaluate it at the same point arises again, we look up the old value in a table we maintain for this purpose instead of repeating the work. The implementation does this.

The most straightforward implementation of memo tables would implement the tables as association lists, and always use type equality to search for a relevant entry in the table. The present implementation does indeed implement the tables as association lists, but searching the tables is slightly more clever. Since type equality can be slow when types have negative arguments, we compare types with negative arguments using the `sameas` field of the `teqopt`; this is essentially the same as using pointer equality, except that if two types

have been found equal while searching pending analysis tables, we use that information when searching the memo tables.

Memoization of refinement types can be turned off by setting the `dont_memoize` flag. This optimization does help; we can run the `CNF` example from Chapter 1 in 22 seconds with the flag clear, and 27 seconds with the flag set.

As a special case, we use a simpler algorithm to memoize types with no negative arguments. In this case the argument to the function in the argument to `RefCon` is always the empty list, so we can omit the work of searching the memo table altogether.

This special case can be turned off by setting the `lazy_refcon` flag. When this is done, these types are memoized with the general-purpose memoizer only. In the `CNF` example from Chapter 1, there are many simple types with no negative arguments, so setting the flag causes an even larger slowdown than `dont_memoize`. This example runs in 22 seconds with this flag clear and 28.5 seconds with the flag set.

Non-constant types are not entered into memo tables or compared with types in memo tables.

7.3 Instantiating Refinement Types

Instantiating refinement types is straightforward when they are represented explicitly. For example, instantiating α to $bool$ in the refinement type $(\alpha \to \alpha) \to \alpha \to \alpha$ yields

$$\begin{array}{ll} (tt \to tt) \to tt \to tt & \wedge \\ (ff \to ff) \to ff \to ff & \wedge \\ (\top_{bool} \to \top_{bool}) \to \top_{bool} \to \top_{bool} & \wedge \\ (\bot_{bool} \to \bot_{bool}) \to \bot_{bool} \to \bot_{bool} & . \end{array}$$

An algorithm for this is straightforward: simply enumerate all refinement type substitutions refining the ML type substitution, apply each of them to the original refinement type, and take the intersection of all the results. Unfortunately, this procedure is slow; the number of refinement type substitutions to consider grows exponentially as a function of the number of type variables to be instantiated. In this section we give an instantiation algorithm that instantiates lazily represented refinement types without enumerating all possible refinement type substitutions. The correctness proof for this algorithm is future work.

The purpose of this section is to make the instantiation algorithm intuitively plausible and to describe it well enough to permit interested people to attempt to prove or disprove soundness. One obstacle to the soundness proof is devising specifications for the various subroutines in the algorithm that are both formal and correct. All specifications below will be informal.

7.3.1 Instantiation Example and Algorithm

The function

$$\texttt{fn f:}\alpha \to \alpha \texttt{ => fn x:}\alpha \texttt{ => f[] (f[] x[])}$$

has the refinement type scheme $\forall(\alpha).(\alpha \to \alpha) \to \alpha \to \alpha$. Call this function `double`. Our example consists of using a lazy representation of refinement types to determine the principal refinement type of `double`$[bool]$ `not[] (true ())`, where the booleans have the usual refinements and `not` is the obvious function mapping booleans to booleans. As the argument in Subsection 4.1.2 on page 226 shows, the correct result is $\top_{bool}$.

To make the explanation simple, we use a simplified version of the lazy refinement type representation introduced above:

```
datatype boolref = TT | FF | Bot | Top
datatype tp = Ground of boolref
  | Later of (tp -> tp)
  | Reftyvar
```

In this datatype, `Ground` constructs refinements of $bool$ in the obvious way. `Later` is a simplified version of the `RefCon` constructor that only applies to function types; `Later` f is the refinement type with the interpretation (as defined in Chapter 2) f. The value constructor `Reftyvar` stands for a refinement of a type variable. It is always clear from context which type variable `Reftyvar` refines.

To conveniently describe types in terms of this datatype, we will need an inverse for `Later`:

```
exception Bug of string
fun now (Later f) = f
  | now _ = raise Bug "now"
```

Using this, we can write something equivalent to the representation of the refinement type of `double` that would arise from the natural type inference algorithm:

```
Later (fn f => Later (fn x => (now f) ((now f) x)))
```

With the exception of the insertions of the `Later`'s and `now`'s, this has the same structure as the definition of `double` itself. This is not surprising since refinement type inference is a form of abstract interpretation and `double` contains no value constructors. Here we assume that the refinement type given for `f` will always refine $\alpha \to \alpha$ and the refinement type given for `x` will always refine α; this implies the type passed for `x` will always be `Reftyvar`. (Later, we will consider an alternative valid refinement type for `double` other than the one that would arise from the natural type inference algorithm.) We can also give the type for `not[]` in this format:

```
let fun notfb TT = FF
      | notfb FF = TT
      | notfb Top = Top
      | notfb Bot = Bot
    fun notfa (Ground x) = (Ground (notfb x))
      | notfa _ = raise Bug "notfa"
    in Later notfa end
```

and the type of `true ()` is `Ground TT`.

The heart of the instantiation algorithm we will describe below only works for types with no negative type arguments. Thus the first step toward determining the type for `double`$[bool]$ `not[] (true ())` is postponing the real work of instantiation until we are left with the problem of instantiating a type with no negative type arguments. This postponment is necessary for the correctness of the algorithm we describe below. An example where the algorithm is incorrect if this step is omitted appears on page 292.

The ML type type of `double`$[bool]$ is $(bool \to bool) \to bool \to bool$, which has negative type arguments. Thus we postpone work; the refinement type generated for `double`$[bool]$ is simply `Later (fn f' => ...)`, where we will fill in the "..." in a moment.

While finding the type for `double`$[bool]$ `not[]`, we will strip the `Later` from the type of `double`$[bool]$ and pass `Later notfa` to the resulting function. The result of this must be a refinement of $bool \to bool$, so we postpone work further by giving this the form `Later (fn x' => ...)`. This implies the refinement type of `double`$[bool]$ must have the form `Later (fn f' => Later (fn x' => ...))`.

While finding the type for `double`$[bool]$ `not[] (true ())`, we will strip the `Later` from the type of `double`$[bool]$ `not[]` and pass `Ground TT` to the resulting function. The result will refine $bool$, which has no negative type arguments; thus we are finished with the stage where we are postponing the real work of instantiation.

Once we have `f'` and `x'`, we will search for the least substitution mapping type variables to refinement types that is consistent with the types `f'` and `x'`. In our example, `f'` is bound to the type of `not[]` as described above and `x'` is bound to `Ground TT`. Given a substitution $r\sigma$, we can convert `f'` and `x'` into refinement types we can pass for `f` and `x` in the before-instantiation type of `double`. We call this process "reshaping".

We will talk about two different reshaping processes. One is called `reshapeab` because it *reshape*s an *a*fter-instantiation refinement type like `f'` into a *b*efore-instantiation refinement type like `f`. Another is called `reshapeba` because it *reshape*s a *b*efore-instantiation refinement type into an *a*fter-instantiation refinement type. These two procedures are mutually recursive. They refer to two global variables: $r\sigma$ is the present substitution of refinement types for type variables, and $m\sigma$ is a fixed substitution of ML types for type variables.

As described above, we are looking for a least $r\sigma$ that is consistent with the types `f'` and `x'`. We perform this search by starting with the least $r\sigma$ and revising it as necessary

until it is consistent with the types `f'` and `x'`. The code below indicates that revisions are necessary by raising an exception; specifically, the exception `TooSmall` $(\alpha,\ r)$ is raised to indicate that $r\sigma$ should be replaced by $r\sigma[\alpha := r]$.

Although we have no formal specification for `reshapeab` and `reshapeba`, we can formally describe a few invariants used in it. Whenever `reshapeab` $r\ t$ is called, $r \sqsubset m\sigma(t)$. When evaluating `reshapeab` $r\ t$ raises no exception, the result refines t. Whenever `reshapeba` $r\ t$ is called, $r \sqsubset t$, and any result refines $m\sigma(t)$.

```
fun reshapeab (Later f) (t1 → t2) =
    Later (fn r => reshapeab (f (reshapeba r t1)) t2)
  | reshapeab r bool = r
  | reshapeab r α =
    if subtypep r rσ(α) mσ(α) then
       Reftyvar
    else
       raise TooSmall (α, joinf r rσ(α) mσ(α))
and reshapeba (Later f) (t1 → t2) =
    Later (fn r => reshapeba (f (reshapeab r t1)) t2)
  | reshapeba r bool = r
  | reshapeba r α = rσ(α)
```

To solve the instantiation problem at hand, we will use `reshapeab` to convert `f'` and `x'` to the `f` and `x` expected in the uninstantiated type for `double`. Then we will use `reshapeba` to convert the value returned from the type for `double` into a refinement of *bool*.

Now we shall apply these algorithms to `f'` and `x'`. For the instantiation problem we have in mind, $m\sigma$ is

$$[\alpha := \mathit{bool}];$$

we will leave $r\sigma$ undetermined for the time being. Evaluating `reshapeab f'` $(\alpha \to \alpha)$ and simplifying yields

```
Later (fn r =>
       if subtypep (notfa (rσ(α))) rσ(α) bool then
          Reftyvar
       else raise TooSmall (α, joinf r rσ(α) bool))
```

and evaluating `reshapeab x'` α and simplifying yields

```
if subtypep (Ground TT) rσ(α) bool then
   Reftyvar
else
   raise TooSmall (α, joinf (Ground TT) rσ(α) bool)
```

We start by assuming that $r\sigma$ is the least possible substitution that refines $m\sigma$, which is

$$[\alpha := \texttt{Ground Bot}].$$

With this assumption, determining `reshapeab x'` α immediately raises the exception

$$\texttt{TooSmall } (\alpha, \texttt{ Ground TT}),$$

so we revise $r\sigma$ to

$$[\alpha := \texttt{Ground TT}].$$

Starting with the new $r\sigma$, we find that `reshapeab f'` α yields

```
Later (fn r =>
       if subtypep (notfa (Ground TT)) (Ground TT) bool then
          Reftyvar
       else raise TooSmall (α, joinf r (Ground TT) bool))
```

and `reshapeab x'` α yields α. Now we pass these two values for `f` and `x` respectively in `(now f) ((now f) x)`; the definition of `f` then raises the exception

$$\texttt{TooSmall } (\alpha, \texttt{ Ground Top}).$$

Thus we revise the substitution to

$$[\alpha := \texttt{Ground Top}]$$

and try again. This time no exceptions are raised, and the value returned by

$$\texttt{(now f) ((now f) x)}$$

is `Reftyvar`. Then we call `reshapeba Reftyvar` α, which yields `Ground Top`, which is our solution.

If we do not postpone as much work as possible, this algorithm gives incorrect results. For example, suppose we want to instantiate α to *bool* in refinement type `Later (fn` x `=>` x`)` interpreted as a refinement of $\alpha \to \alpha$. If we do not postpone any work, we start with the assumption $r\sigma = [\alpha := \texttt{Ground Bot}]$ and end with the same substitution. The result from instantiation is `reshapeba (Later (fn` x `=>` x`))` $(\alpha \to \alpha)$, which simplifies to

$$\texttt{Later (fn } x' \texttt{ => reshapeba (reshapeab } x' \texttt{ } \alpha) \texttt{ } \alpha)$$

which in turn simplifies to

```
Later (fn x' => if subtypep x' (Ground Bot) bool then Ground Bot else
                    raise TooSmall (α, x')).
```

Thus instantiation returns a refinement type that will raise `TooSmall` under some conditions; this is clearly malformed.

To finish the instantiation algorithm, we will describe the code wrapped around the definitions of `reshapeab` and `reshapeba`, and we will describe and fix one situation where the above code is incorrect. The resulting procedure has no known bugs but no correctness proof.

There are two parts to the remaining code: the procedure for postponing work until we have no negative type arguments and the loop searching for an appropriate $r\sigma$. Both of these are straightforward; we will present postponing the work first, since it is outermost. In the following definition, the call `inst` r $m\sigma$ t instantiates the refinement type r according to the substitution $m\sigma$ mapping type variables to ML types, under the assumption that r refines t. In the code below, we assume that `mapsubst` f σ constructs a substitution with the same domain as σ and for all α in that domain, (`mapsubst` f σ)(α) = $f(\sigma(\alpha))$. The `botfn` function was introduced on page 118 for computing the least refinement of any ML type. We will fill in the definition of the function `looper` later.

```
fun inst r mσ t =
   let fun instargs args argtys r (t1 → t2) =
           Later (fn arg => instargs (arg :: args) (t1 :: argtys) r t2)
         | instargs args argtys r t =
           let fun looper rσ = ...
           in
              looper (mapsubst botfn mσ)
           end
   in
      instargs [] [] r t
   end
```

The above code is straightforward; it simply accumulates refinement types and ML types in the $args$ and $argtys$ arguments until the refinement type does not refine a functional type, and then it calls `looper` with a suitable initial value for $r\sigma$.

Now we can give a definition of the function `looper` that iterates to find the least substitution consistent with the constraints. This definition has the free variables $args$, $argtys$, r, and t. In the code below, `rev` is the standard function for reversing lists.

```
fun looper rσ =
   let exception TooSmall of string * tp
       fun reshapeab ... = ...
       and reshapeba ... = ...
       fun call (Later f) (arg1 :: argrest) (at1 :: atrest) =
           call (f (reshapeab arg1 at1)) argrest atrest
         | call x [] [] = reshapeba x t
         | call _ _ _ = raise Bug "call"
       in
         (call r (rev args) (rev argtys)
          handle
          TooSmall (var, tp) =>
              looper (rσ[var := tp])
       end
```

This code is fairly straightforward; it uses the stored argument lists to repeatedly call the uninstantiated refinement type, changing $r\sigma$ as indicated by the `TooSmall` exceptions until $r\sigma$ is a usable substitution.

The symmetry between `reshapeab` and `reshapeba` is pleasing, and the algorithm specified above seems to work if all refinement types are generated by a natural algorithm starting with expressions without any explicit refinement type declarations, and all arguments are used. Unfortunately, making an algorithm that appears to work in general breaks the symmetry. For example, this `let` statement

```
let foo = Λ(α).fn f:α→α => fn x:α => x[]
in ... end
```

will add the refinement type scheme

$$\forall(\alpha).(\alpha \to \alpha) \to \alpha \to \alpha$$

to the environment before it typechecks the expression within the scope of the `let` statement. This is the same as the refinement type scheme resulting from the `double` example above, except now the representation of the refinement type that results from the natural algorithm is

```
Later (fn f => Later (fn x => x)).
```

Since `f` is never used, the above instantiation algorithm will not inspect the type of `f`. This is clearly wrong; since the refinement type of `foo` is the same as the refinement type of `double`, instantiating the type of `foo` should pull the same information out of `f` that instantiating the refinement type of `double` does.

Suppose we used the present algorithm to determine the type of

```
foo[bool] not[] (true ()).
```

The instantiation algorithm would start with $r\sigma$ equal to

$$[\alpha := \texttt{Ground Bot}].$$

As in the earlier example, this would be observed to be inconsistent with the type of `true ()`, so we would revise $r\sigma$ to

$$[\alpha := \texttt{Ground TT}].$$

Here the resemblance to the earlier example ends, because unlike that example the type of `not[]` is never examined; the algorithm is finished and the result is `Ground TT`.

Intuitively, it seems plausible that the problem is that the final substitution is not consistent with the type of `not[]`; in other words, if we use `reshapeab` to de-instantiate the type of `not[]` with the final $r\sigma$, the resulting type raises an exception for all inputs. Since the interpretation of a refinement is always a monotone function, it will fail for all inputs if and only if it fails for the least input. Thus we can detect this problem by passing the least refinement of the input ML type to the function and discarding the result; any problems will be dealt with as a consequence of the resulting `TooSmall` exception. Thus we rewrite the function case of `reshapeab` as follows:

```
fun reshapeab (Later f) t1 -> t2 =
    (reshapeab (f (botfn (applysubst mσ t1))) t2;
     Later (fn arg => instargs (arg :: args) (t1 :: argtys) r t2))
```

With this rewritten case, the algorithm has no known bugs. Assembling the pieces of code appearing in this chapter yields the completed algorithm in Figure 7.1.

7.3.2 Memoizing Instantiation

Every variable is instantiated before it is used, although the instantiation is often trivial. This makes memoizing instantiation very important. If we do not do this, then each type is created anew every time a variable is referenced; these newly created types have empty memo tables, so unmemoized instantiation undoes many of the other memoization optimizations. The implementation normally memoizes instantiation; the flag `dont_memoize_inst` can be set to turn this off.

As implemented, the instantiation algorithm quickly deals with nonpolymorphic types by using a special case. Thus the `CNF` example above cannot be used to illustrate this optimization. We can illustrate it by using a simple polymorphic type, such as polymorphic lists, even if we make no interesting use of the polymorphism. For example, if we distinguish even length lists from odd length lists and empty lists from nonempty lists, then a simple function for appending lists:

```
fun inst r mσ t =
   let fun instargs args argtys r (t1 → t2) =
           Later (fn arg => instargs (arg :: args) (t1 :: argtys) r t2)
         | instargs args argtys r t =
           let fun looper rσ =
               let exception TooSmall of string * tp
                   fun reshapeab (Later f) t1 → t2 =
                       (reshapeab (f (botfn (applysubst mσ t1))) t2;
                        Later (fn arg =>
                                  instargs (arg :: args) (t1 :: argtys) r t2))
                     | reshapeab r bool = r
                     | reshapeab r α =
                       if subtypep r rσ(α) mσ(α) then
                          Reftyvar
                       else
                          raise TooSmall (α, joinf r rσ(α) mσ(α))
                   and reshapeba (Later f) (t1 → t2) =
                       Later (fn r => reshapeba (f (reshapeab r t1)) t2)
                     | reshapeba r bool = r
                     | reshapeba r α = rσ(α)
                   fun call (Later f) (arg1 :: argrest) (at1 :: atrest) =
                       call (f (reshapeab arg1 at1)) argrest atrest
                     | call x [] [] = reshapeba x t
                     | call _ _ _ = raise Bug "call"
                   in
                     (call r (rev args) (rev argtys)
                      handle
                      TooSmall (var, tp) =>
                          looper (rσ[var := tp])
                   end
           in
              looper (mapsubst botfn mσ)
           end
   in
      instargs [] [] r t
   end
```

Figure 7.1: Instantiation algorithm.

```
fix ap:α list → α list → α list =>
   fn x:α list => fn y:α list =>
   case x of
      cons => fn hdtl:α * α list =>
         cons (elt_1_2 hdtl, ap (elt_2_2 hdtl) y[])
    | nil => fn x:tunit => y[]
   end:α list
```

has a refinement type that is an intersection of 49 components. Computing this type takes 7.3 seconds with `dont_memoize_inst` turned off, or 60 seconds with `dont_memoize_inst` turned on.

7.4 Analyzing Rectype Declarations

Most of the code for analyzing `rectype` declarations is either obvious or an implementation of some algorithm in Chapter 3. A brief description of the known shortcomings of the implementation follows.

No attempt has been made to enforce Assumption 5.29 (Predefined Intersection Distributivity) on page 259. Sometimes this assumption does not hold and the implementation behaves strangely; for example, with the usual declaration for *bool* and this declaration:

```
datatype α d = C of bool → α
rectype α z = C (tt → α) | C (ff → α);
```

the implementation infers that `C ((fn x => x) ◁ (tt → tt))` has the principal type $tt\ z$ and

```
C ((fn x => x) ◁ (ff → ff))
```

has the principal type $ff\ z$, but it also infers that

```
C ((fn x => x) ◁ (ff → ff ∧ tt → tt))
```

has the principal type $\top_{bool}\ z$. This means that the behavior inferred for `C` is not monotone, a serious bug. Fixing this by inferring a different behavior for `C` seems more satisfying than fixing it by outlawing declarations similar to this one, but the best way to do this is not immediately clear. In this example, the instantiation algorithm is misbehaving in circumstances where Predefined Intersection Distributivity is false; thus it is reasonable to guess that any soundness proof for the instantiation algorithm will use Predefined Intersection Distributivity.

When we infer the predefined splitting relation from the `rectype` declaration, we assume without proof that it suffices to consider exactly one most informative principal split of each constructor. The implementation uses a brute force search to find all of the

"plausible" principal splits of each constructor; in this context a proposed split is plausible if no two types in it are comparable, and all of the fragments are less than the constructor of which they are proposed fragments. Then another brute force search lists fixed points of the inference system in Figure 3.8 on page 207 where we assume each new refinement type constructor has at most one split in the fixed point. We assume without proof that the most informative of these is an appropriate predefined splitting relation.

7.5 Differences Between Implementation and Theory

There are a few differences between the implementation and the theory that do not fit neatly into any of the topics listed above.

We treat `case` statements where the case object is a variable specially. For example, suppose we have the declarations

```
datatype maybe = true of tunit | false of tunit | maybe of tunit
rectype tt = true (tunit)
    and ff = false (tunit)
    and tf = true (tunit) | false (tunit)
datatype forget = C of maybe
```

Then the best type for `x` from

```
val x = case C (true ()) of C => fn y => y end:maybe
```

is $\top_{maybe}$. Reading the type system strictly, the statement

```
case x of
    true => fn _ => false ()
  | false => fn _ => x
  | maybe => fn _ => false ()
end:maybe;
```

has the best type $\top_{maybe}$ because in the `false` case, the type of the variable `x` is still $\top_{maybe}$, since `case` statements do not affect variable bindings. This surprises many users because the `case` statement obviously always returns `false ()`. To eliminate the surprise, when the case object is a variable (`x` in the example), the implementation binds that variable to a better type while analyzing each branch of the case statement. The better type is computed by applying the constructor for each case to the inferred type of its argument; in this example, the constructor is `false` and the inferred type of the argument is the unique refinement of *tunit*, so `x` is bound to the type *ff* within the scope of the `false` branch of the `case` statement.

Another practical inaccuracy in the implementation is that types appearing at the top level are not split. For example, assuming the usual definitions of the booleans, `not`, and `or`, splitting causes the expression

$$(\texttt{fn x => (or x (not x))})\ ((\texttt{true ()}) \lhd \top_{bool})$$

to get the type tt. However, if the declaration

$$\texttt{val x = (true ())} \lhd \top_{bool}\texttt{;}$$

is followed by the expression

$$\texttt{or x (not x),}$$

then the latter expression only gets the type $\top_{bool}$. We do this because the implementation of splitting requires reanalyzing the entire scope of the binding for each fragment of the split; if the scope is the entire future history of the type checker, then we cannot afford to do this.

Chapter 8

Conclusion, Critical Evaluation, and Future Work

Refinement type inference shows signs of being a useful type inference system. The types have an intuitively appealing meaning, type inference can be described with readable inference rules, type inference provably has some useful properties, and a working implementation exists.

As with any work of this size, this one has shortcomings. Some of the shortcomings represent tradeoffs made to ensure that refinement type inference is efficiently decidable. Other shortcomings could be remedied by experimenting, adding new language features, proving more theorems, or by improving the implementation.

8.1 Tradeoffs Made for Tractable Type Inference

There are numerous situations where a program has a property that can be expressed as a refinement type, but refinement type inference cannot infer as strong a type as one would like.

Refinement type inference only makes the distinctions specified by the programmer in `rectype` declarations. Even if a true property of a program can be described in terms of those distinctions, if one must use other distinctions to infer this, refinement types cannot infer that the property is true. For example, consider the declarations from Chapter 1 that distinguish lists of length zero, one, and two or more from each other:

```
datatype α list = nil | cons of α * α list
rectype α empty = nil
    and α singleton = cons (α, nil)
    and α long = cons (α, cons (α, α ⊤list))
    and α ⊥list = bottom (list)
```

Then the function

```
fn x => case cons (x, cons (x, nil)) of
          cons (_, cons (_, nil)) => cons (x, nil)
        | _ => nil
```

will always return a list of length one, but it will not have the refinement type $\alpha \rightarrow \alpha\ singleton$ because we can only determine that the function will always return a list of length one by recognizing a list of length exactly two, and we have assumed that no `rectype` declaration has been made that will distinguish lists of length exactly two. It is easy to express this distinction as a `rectype` statement:

```
rectype α twolist = cons (α, cons (α, nil))
```

In general, `rectype` statements are descriptions of regular tree automata [GS84], and sets of values that are not recognizable by a finite tree automaton cannot be described with a `rectype` statement. For example, we can make the usual distinction among the booleans

```
datatype bool = true of tunit | false of tunit
rectype tt = true (runit)
    and ff = false (runit)
```

and write a function to test whether two lists have the same length:

```
fun samelength (cons (x, tlx)) (cons (y, tly)) = samelength tlx tly
  | samelength nil nil = true
  | samelength _ _ = false
```

With this definition, for any list l we know that `samelength` l l returns `true`, but we cannot declare any finite set of distinctions within *list* to cause `samelength` to have the refinement type $\top_{list} \rightarrow \top_{list} \rightarrow tt$. The problem here is that the infinite set of possible lengths cannot be encoded in the state of a finite tree automaton. Similarly, refinement type inference cannot reason about closed expressions in a representation of the lambda calculus because the infinite number of possible sets of bound variables cannot be encoded in the state of a finite tree automaton.

Another shortcoming is that refinement type inference does not know when a function is deterministic and unaffected by side effects. Thus, if l is some list, we will not be able to infer that the expression

```
if samelength l l then true else not (samelength l l)
```

has the refinement type tt. If refinement types were able to use the information that `samelength` is deterministic and does not use side effects, it could infer that the `if`

statement has the type tt even if it could not infer that `samelength` l l can be given the type tt.

A different shortcoming stems from the fact that refinement type inference is defined in terms of expressions with explicit ML types, but the programmer writes expressions with implicit ML types. In general, a term with implicit ML types may correspond to multiple terms with explicit ML types. To predict the behavior of refinement types, the programmer needs to know which one of these the compiler will select. Fortunately, the prototype implementation (and every Standard ML implementations that uses the simplest algorithm) always selects the explicitly typed term containing the most general types.

The need to insert a `rectype` declaration before refinement type inference provides more information than ordinary ML type inference can also be regarded as a shortcoming. However, it is hard to imagine doing without `rectype` declarations. In the normal case, refinement type inference will be used to find errors in a recently modified program. Analyzing the program to automatically find the important distinctions to make is likely to be hopeless when the program is incorrect. The often-suggested option of omitting `rectype` statements and instead automatically creating one refinement containing each value constructor is unworkable; refinement type inference will give some information in this case, but the information will rarely be useful. For example, it would not have been useful for any of the examples in the introduction.

8.2 Experience Yet to Be Gained

Sometimes it is not clear which distinctions need to be made in a `rectype` declaration to get the desired conclusion. In the function

```
fun lastcons (last as cons (hd, nil)) = last
  | lastcons (cons (hd, tl)) = lastcons tl
```

we need to use a `rectype` declaration to distinguish lists of length two to be able to infer that `lastcons` has the type $\alpha\ \top_{list} \rightarrow \alpha\ singleton$. If we give the type of lists of length two the name $\alpha\ long$, the following argument shows why we need to distinguish $\alpha\ long$ to get the best type for `lastcons`: All values of type $\alpha\ \top_{list}$ are in one of $\alpha\ empty$, $\alpha\ singleton$, or $\alpha\ long$. Each of these cases falls squarely into one of the branches of the definition of `lastcons`: if the argument is of type $\alpha\ long$, then we will always get to the recursive call `lastcons`; if the argument is of type $\alpha\ singleton$, then we return the argument; and if the argument is in $\alpha\ empty$, then we raise an exception because of a missing case.

If we omit the declaration of $long$, then we can no longer say that all values of type $\alpha\ \top_{list}$ are in one of several smaller types. If the argument to `lastcons` has type $\alpha\ \top_{list}$, then the first case of `lastcons` is reachable, and we return `last`, which is the argument to `lastcons` and therefore has the type $\alpha\ \top_{list}$. Thus, from the viewpoint of type inference, `lastcons` appears to be able to return a value of $\alpha\ \top_{list}$. This could be fixed by a more

careful understanding of how patterns bind to variables that gives `last` the type $\alpha\ singleton$ in this case, or, as we mentioned in the previous paragraph, it could be fixed by adding the refinement type `long`.

As larger programs are checked with refinement type inference, the programmer will become more experienced, but there will also be greater opportunity for surprising scenarios like the one described in the previous paragraph to happen. It is unclear whether this process will lead to sufficiently rare surprises in the long run; more experience is necessary.

More experience is also necessary to determine how fast and how useful refinement type inference will be for large programs. The largest program run through the type checker so far is the conjunction normal form example in Section 1.2, which is only 50 lines of SML code.

8.3 Future Work in Language Design

The correct interaction between refinement types and signatures is not clear. For example, suppose we have a structure `List` that implements lists and operations on them such as `append`, and suppose another structure uses `List` and uses a statement to make a distinction between empty and nonempty lists. Getting the best possible refinement type for `append` in the second structure requires re-analyzing the code in the context of the added `rectype` statement; assuming that type inference respects the privacy of `List`, re-analyzing the code will require repeating the code in the second structure, which is poor software engineering.

Another option would be to allow `List` to declare the implementation of `append` in its signature to give type inference permission to re-analyze `append` as necessary when new `rectype` declarations are added. Putting expressions in signatures is a big change to SML; more work is necessary to determine whether this is worthwhile.

Some data types such as *string* and *int* are predefined rather than declared with a `datatype` statement. It makes sense to have refinements of these; for example, we could imagine distinguishing positive integers, negative integers, and zero from each other. However, this will require a declaration other than a `rectype` statement, since `rectype` statements rely upon having a finite number of constructors for each data type. It may be worthwhile to find some other way to declare refinements of predefined data types.

If we omit the declaration of $long$ in the *list* example, `lastcons` does not get the right type. Adding $long$ brings about the right result because we can then infer that all values in $tt\ list$ are in one of the types $tt\ empty$, $tt\ singleton$, or $tt\ long$; without $long$, there are values such as `cons (true (), cons (true (), nil))` that are in $\alpha\ list$ but are not in any smaller type. The example with *cnf* is not analogous; we can infer an accurate type for `toCnf` without having a refinement type that represents all boolean expressions that are not in CNF. In general, small variations in the `rectype` declaration have a subtle effect on the outcome of refinement type inference. Perhaps some useful rules of thumb will arise from

experiments with larger programs.

Every visible `rectype` declaration increases the number of cases that refinement type inference must examine and therefore slows down type inference. Thus it is very important to have good mechanisms for restricting the scope of a `rectype` declaration to a small extent of code. The problem with this is that it is not immediately obvious what should happen when we leave the scope of a `rectype` declaration. For example, suppose we have the declarations of $bool$, tt, and ff on page 301 and consider the statement

```
let datatype α list = nil | cons of (α * α list)
    val x = cons (true, nil)
in
   let rectype α ev = nil | cons (α * α od)
           and α od = cons of (α * α ev)
   in
       (x ◁ tt od;
        cons (true, nil))
   end
end
```

In Standard ML, the type constructor $list$ becomes anonymous once we leave the scope of the outer `let` statement; this means that the type still exists, but it cannot be named in type declarations. Should the same happen to the recursive type constructors ev and od when we leave the scope of the outer `let` statement? The practical and theoretical consequences of this have not been explored.

It would be better if the specification of the meaning of `rectype` declarations in Chapter 3 were more declarative. Also, because many properties of regular tree sets are effectively decidable, it is possible in principle to do perfect reasoning about `rectype` declarations that do not mention function types. An example of this weakness of `rectype` declarations as currently specified is on page 193. It would be more satisfying to have a specification of the meaning of `rectype` statements that was as accurate as possible in that case.

8.4 Future Theoretical Work

The soundness theorem in Chapter 2 states that if we evaluate a closed expression to get a value, then the value has any refinement type the closed expression did. It does not immediately follow that every time we evaluate a subexpression of the form $e \lhd r$, all values computed for e actually had the type r. A more ambitious soundness proof would show this.

A version of refinement type inference that deals with imperative features such as references exists and has a soundness proof, but has not yet been written up.

The type inference system in Figure 3.8 for deriving the splitting relation from `rectype` statements is not directly implementable. The prototype implementation does something that seems to work well in practice, but it needs to be verified.

Likewise, the instantiation algorithm used by the implementation seems to work well in practice but it needs to be verified.

8.5 Future Implementations

The present implementation uses memo tables in many places to improve performance. These memo tables are all implemented as lists; profiling shows that the implementation spends 80% of its time searching these lists. This could be sped up dramatically by using arrays and an appropriate hashing scheme.

The implementation does not use true SML syntax. For instance, the syntax for defining functions is separate from the syntax for destructuring data types. Also, constant value constructors like `true` are not permitted; instead, every value constructor takes one argument, so the best we can do is `true ()`.

The theory allows for four ways a type variable can appear as an argument to a polymorphic data type constructor: positive, negative, ignored, or mixed. We only implement positive and negative. Ignored arguments are treated as though they are positive, and mixed ones generate an error. Type variables appearing in references behave as though they are mixed, so the prototype does not implement references either.

Typically the refinement type of a function is very large, and we are only interested in a small part of it. For example, we can verify that `toCnf` has the type $\top_{boolexp} \rightarrow cnf$ fairly quickly, but there are several refinements of *boolexp*, so it takes a while to print the entire type of `toCnf`. The implementation needs to be more careful not to print these expensive-to-compute types.

For a similar reason, when a refinement type error occurs, it is difficult to discover why. A good approach to this might be to provide an interactive dialogue so the user can ask the type inference engine questions about how the error occurred. This has been explored for ML [Wan86].

Refinement type inference can in principle be used to make code more efficient. For example, in the expression

```
case lastcons y of
   cons (x, nil) => print x
```

refinement type inference could guarantee to the compiler that the value returned by `lastcons` will be a `cons` cell, so the `case` statement does not have to verify this.

Bibliography

[AC90] Roberto M. Amadio and Luca Cardelli.
Subtyping recursive types.
Research Report 62, Digital Systems Research Center, Palo Alto, California, August 1990.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[Bar80] H. P. Barendregt.
The Lambda-Calculus: Its Syntax and Semantics.
North-Holland, 1980.

[Car87] Luca Cardelli.
Basic polymorphic typechecking.
Science of Computer Programming, 8:147–172, 1987.

[CDCV80] Mario Coppo, Mariangiola Dezani-Ciancaglini, and B. Venneri.
Principal type schemes and λ-calculus semantics.
In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 535–560. Academic Press, London, 1980.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn.
A simple applicative language: Mini-ML.
In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[CG92] M. Coppo and P. Giannini.
A complete type inference algorithm for simple intersection types.
In J.-C. Raoult, editor, *17th Colloquium on Trees in Algebra and Programming, Rennes, France*, pages 102–123, Berlin, February 1992. Springer-Verlag LNCS 581.

[CR36] Alonzo Church and J.B. Rosser.
Some properties of conversion.
Transactions of the American Mathematical Society, 36(3):472–482, May 1936.

[dB72] N. G. de Bruijn.
Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem.
Indag. Math., 34(5):381–392, 1972.

[Dix88] Alan J. Dix.
Finding fixed points in non-trivial domains: Proofs of pending analysys and related argorithms.
Technical Report YCS 107, University of York Department of Computer Science, Heslington, York, YO1 5DD, England, 1988.
An undated addendum has been written that describes significant improvements.

[DM82] Luis Damas and Robin Milner.
Principal type schemes for functional programs.
In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM SIGPLAN/SIGACT, 1982.

[DZ92] Philip W. Dart and Justin Zobel.
A regular type language for logic programs.
In Frank Pfenning, editor, *Types in Logic Programming*, chapter 5, pages 157–187. MIT Press, Cambridge, Massachusetts, 1992.

[FM89] You-Chin Fuh and Prateek Mishra.
Polymorphic subtype inference: Closing the theory-practice gap.
In *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain*. Springer-Verlag, March 1989.

[FM90] You-Chin Fuh and Prateek Mishra.
Type inference with subtypes.
Theoretical Computer Science, 73:155–175, 1990.

[GS84] Ferenc Gécseg and Magnus Steinby.
Tree Automata.
Akadémiai Kiadó, Budapest, 1984.

[Har86] Robert Harper.
Standard ML.
Technical Report ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University, March 1986.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin.
A framework for defining logics.
Journal of the Association for Computing Machinery, 40(1):143–184, January 1993.

[HL94] Robert Harper and Mark Lillibridge.
A type-theoretic approach to higher-order modules with sharing.
To appear in POPL '94., 1994.

[HM94] Chris Hankin and Daniel Le Métayer.
Deriving algorithms from type inference systems: Application to strictness analysis.
In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland*, pages 202–212. ACM, January 1994.

[HMM+88] Robert Harper, Robin Milner, Kevin Mitchell, Nick Rothwell, and Don Sannella.
Functional programming in Standard ML.
Notes to a five day course given at the University of Edinburgh, April 1988.

[HP91] Robert Harper and Benjamin Pierce.
A record calculus based on symmetric concatenation.
In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 131–142, Orlando, Florida, January 1991.

[Jag89] Nigel Jagger.
An inductive approach to finding fixpoints in abstract interpretation.
Fourth IEEE Region 10 International Conference, pages 1059–1064, 1989.

[Jat89] Lalita A. Jategaonkar.
ML with extended pattern matching and subtypes.
Master's thesis, Massachusetts Institute of Technology, August 1989.

[JM86] Neil D. Jones and Alan Mycroft.
Data flow analysis of applicative programs using minimal function graphs: Abridged version.
In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach*, pages 296–306. ACM, January 1986.

[JM88] Lalita A. Jategaonkar and John C. Mitchell.
Ml with extended pattern matching and subtypes (preliminary version).
In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 198–211. ACM, July 1988.

[KTU89] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn.
Type-checking in the presence of polymorphic recursion.
To appear in TOPLAS, October 1989.

[LW91] Xavier Leroy and Pierre Weis.
Polymorphic type inference and assignment.
In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando*, pages 170–181. ACM, January 1991.

[Mac88] David B. MacQueen.
An implementation of Standard ML modules.
In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, Utah*, pages 212–223. ACM Press, July 1988.

[Mil78] Robin Milner.
A theory of type polymorphism in programming.
Journal of Computer and System Sciences, 17:348–375, August 1978.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov.
Uniform proofs as a foundation for logic programming.
Annals of Pure and Applied Logic, 51:125–157, 1991.

[MP91] Spiro Michaylov and Frank Pfenning.
Natural semantics and some of its meta-theory in Elf.
In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[MT91a] Robin Milner and Mads Tofte.
Co-induction in relational semantics.
Theoretical Computer Science, 87:209–220, 1991.

[MT91b] Robin Milner and Mads Tofte.
Commentary on Standard ML.
MIT Press, Cambridge, Massachusetts, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, Cambridge, Massachusetts, 1990.

[Myc84] Alan Mycroft.
Polymorphic Type Schemes and Recursive Definitions, pages 217–228.
International Symposium on Programming. Springer-Verlag, New York, 1984.
LNCS 167.

[PE88] Frank Pfenning and Conal Elliott.
Higher-order abstract syntax.
In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988.

[Pfe92] Frank Pfenning, editor.
Types in Logic Programming.
MIT Press, Cambridge, Massachusetts, 1992.

[Pie91a] Benjamin Pierce.
Programming with intersection types, union types, and polymorphism.
Technical Report CMU-CS-91-106, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1991.

[Pie91b] Benjamin C. Pierce.
Programming with Intersection Types and Bounded Polymorphism.
PhD thesis, School of Computer Science, Carnegie Mellon University, December 1991.
Available as Technical Report CMU–CS–91–205.

[Ré89] Didier Rémy.
Typechecking records and variants in a natural extension of ML.
In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–87, Austin, Texas, January 1989. ACM.

[Rey88] John C. Reynolds.
Preliminary design of the programming language Forsythe.
Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[Tof87] Mads Tofte.
Operational Semantics and Polymorphic Type Inference.
PhD thesis, Department of Computer Science, Edinburgh University, 1987.

[Tof88] Mads Tofte.
Type inference for polymorphic references.
Superseded by a version published in Information and Computation, vol 89, number 1, pages 1-34, November 1990., October 1988.

[TZ91] Rodney Topor and Justin Zobel.
Operations on regular term grammars.
To appear in Acta Informatica., June 1991.

[Wan86] Mitchell Wand.
Finding the source of type errors.
In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 38–43. ACM Press, January 1986.

[YFS92] Eyal Yardeni, Thom Fruehwirth, and Ehud Shapiro.
Polymorphically typed logic programs.
In Frank Pfenning, editor, *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[You89] Jonathan Hood Young.
The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.
PhD thesis, Yale University, May 1989.

Index

$\succeq$, 168
$\equiv$ (equivalence for refinement types), 36
$\times$ (cross product of tuples), 117, 206
|, 166, 168
$\sqsubset$ (refines), 65, 177
<: (ASCII version of coercion operator), 280
$\lhd$ (coercion operator), 13, 273, 304
$\asymp$ (splitting), 46, 65, 93, 253, 297, 299, 305
$\leq$ (subtyping for refinement types), 65, 244
$\approx$ (equivalence for generalized pairs), 247
$\approx$ (equivalence for generalized refinement types), 106
$\preceq$ (subtyping for generalized pairs), 247
$\preceq$ (subtyping for generalized refinement types), 106
$\triangle$ Elim Sub, 107-109, 125, 248
$\triangle$ Intro Sub, 107, 110, 125, 129, 248
$\triangle$ (intersection for generalized pairs), 248
$\triangle$ (intersection for generalized refinement types), 107, 118
$\triangle$`fn`, 117-118, 121, 263
$\sqcup$ (least upper bound of refinement types), 128
$\wedge$ (intersection for refinement types), 3, 16, 18, 30, 211
$\wedge$ (intersection for vectors), 242
& (intersection for recursive types), 165-166, 168, 170, 211

$\overline{\alpha}\{i\}$, 228
ABS-RECVALUE, 180-182, 192, 204-205, 209, 222
ABS-SEM, 24, 100
abstract declaration, well-formed, 177-178
abstract declaration, 169-170, 176-178
abstract interpretation, 3, 282-285
abstract syntax, higher-order, 230
ABS-TYPE, 60-61, 71, 75, 77, 83, 85-86, 89, 95-96, 101, 104, 146, 152, 234
ABS-VALID, 27-28, 71, 74, 77, 232
ALG-ARROW-RECSUB, 195
ALG-NEW-ENV-EMPTY, 186, 188-189
ALG-NEW-ENV-RECSUB, 195
ALG-NEW-INFER-EMPTY, 186-190
ALG-NEW-INFER-RECSUB, 195
ALG-OLD-EMPTY, 186, 188-190
ALG-OLD-RECSUB, 195
algorithm, instantiation, 305
algorithmic emptyness for recursive types ($D; S \vdash r$ alg-empty), 186
Algorithmic Emptyness Strengthening, 188
algorithmic subtyping for recursive types ($D; S \vdash nr \leq nk$), 195
ALG-REC-TUPLE-EMPTY, 186-187, 189-190
ALG-TUPLE-RECSUB, 195
`allrefs`, 117-118, 121, 263, 286
analysis, pending, 281-282, 284
`And`, 6
AND-ELIM-L-SUB, 35-37, 40, 43, 59, 84, 109, 116, 121, 129, 153, 155, 245, 267
AND-ELIM-R-SUB, 35-37, 40, 43, 59, 84, 109, 116, 121, 129, 153, 155, 245, 267
AND-ELIM-TYPE, 59

and-intro-$\overset{\text{def}}{\leq}$, 34, 216, 262
AND-INTRO-SUB, 35-37, 40, 42-44, 84, 109, 113, 139, 245, 267
AND-INTRO-TYPE, 59-61, 67, 69, 75, 80-81, 83, 86-89, 93-94, 99-100, 116, 138-139, 146-148, 191, 259, 262
AND-RECREFINES, 177, 197-198, 216
AND-RECVALUE, 180-181, 191-192, 204-206, 208-210, 221-222
AND-REF, 31-32, 36-37, 39-40, 43, 70, 244
`anysplit`, 126
`append`, 303
APPL-SEM, 24, 92, 99-100, 104, 231
APPL-TYPE, 60, 71, 75, 78, 89, 96, 99, 101-102, 104-105, 147, 154, 222
APPL-VALID, 27-28, 72, 77
applying a substitution, 8
`apsubst`, 8
Arbitrary Constructor Subtyping, 46, 250
arguments, ignored type, 241, 270
arguments, mixed type, 240, 270
arguments, negative type, 240, 270
arguments, positive type, 240, 270
arguments, type, 240-241, 270
arity(tc), 241
arrow ($\rightarrow$), 241
Arrow Does Not Split, 254
ARROW-AND-ELIM-SUB, 34-35, 38, 44, 85, 110, 112, 244
ARROW-RECREFINES, 177, 197
ARROW-RECSPLIT, 206-207, 209
ARROW-RECSUB, 194, 197-199, 202-204
ARROW-REF, 31-32, 38-39, 72, 111, 244
ARROW-SUB, 34-35, 38, 44, 71, 79, 85, 110, 153, 197, 244
ASCII version of coercion operator (`<:`), 280
association lists, 285
atom, 8
automaton, regular tree, 169, 194, 301

bitstr, 17, 65
bitstrings, 65
$\bot_{blist}$, 167
$blist$, 167, 172-173, 177, 207
BOGUS, 253
$\top_{bool}$, 7, 15
$bool$, 7, 15, 174, 224, 226
boolean expressions, 6-7
booleans, 15
$\top_{boolexp}$, 7
boolexp, 6
`boolmap`, 283
`boolref`, 289
`botfn`, 117-118, 121, 134, 293
`bottom` tc, 166-168, 173-174, 176-178
Bound on Argument to i Gives Bound on i, 111, 157, 250, 265, 269
bound variables, 230
de Bruijn indices, 230

C, 114
capture, type variable, 229-230
case, missing, 6-7
Case Statement Body, 265, 269
`case` statements, type inference for, 68
`case` statements, 243, 257, 259, 263, 265, 298
CASE-SEM, 24-25, 102
CASE-TYPE, 29, 60, 63, 67-69, 72, 75, 78-79, 89, 97, 102, 149, 156, 254-255, 257, 259, 265, 268
CASE-VALID, 27-28, 63, 78, 243
catch-all refinement type $\top_t$, 3
claims of this thesis, 12
classes, equivalence, 106
`clearflag`, 276
closed expressions, 22
closure of D ($\overline{D}$), 179, 212
CNF (conjunctive normal form), 8-9, 288, 295
cnf, 8
coercion operator ($\lhd$), 13, 273, 304
coercion operator, ASCII version (`<:`), 280

co-induction, 170, 182, 190, 192, 196, 199-201, 204, 210, 220
compatibility between ML and refinement type inference, 68
components, 111
computing principal splits, 126
concrete syntax, 275, 305
conjunctive normal form (CNF), 8-9, 288, 295
`cons`, 1-2
constant refinement types, 281, 286
CONSTR-SEM, 24, 101
CONSTR-TYPE, 60, 67, 72, 78, 86-87, 89-90, 96-97, 101-102, 148, 155, 221, 254, 257, 259, 261-262
Constructor And Introduction, 67, 87, 136-137, 219, 259, 266
Constructor Argument Strengthen, 67, 87, 148, 155, 212, 220, 259, 262, 266
Constructor mtor Consistent, 76, 78
Constructor Result Weaken, 67, 86, 212, 220, 259, 261, 266
Constructor Type Refines, 65, 72, 79, 155, 217
Constructors have Unique ML Types, 15, 28, 72, 155, 215-216, 241
constructors, ML type (`mlconstructor`), 277-278
constructors, polymorphic refinement type, 240
constructors, properties of, 64
constructors, type, 223
constructors, value, 242
CONSTR-VALID, 27-28, 72, 78, 243
contexts, 172

$\underline{D}$, 176
$\overline{D}$ (closure of D), 179, 212
$D \vdash nr \leq nk$ (subtyping for recursive types), 169, 193-194
$D \vdash nr \asymp s$ (splitting for recursive types), 206-207
$D \vdash nr \sqsubset t$, 177
$D \vdash r$ empty (emptyness for recursive types), 185
$D; S \vdash nr \leq nk$ (algorithmic subtyping for recursive types), 195
$D; S \vdash r$ alg-empty (algorithmic emptyness for recursive types), 186
$D \vdash v \in nr$ (membership of a value in a recursive type), 180
$\widehat{D}$ (weakened closure of D), 213
Damas-Milner type inference, 63
de Bruijn indices, 230
decidability of refinement type inference, 3, 116, 263
decidable ML type inference, 64
Deciding Refinement Types, 135
deciding splits, 126
deciding subtyping, 117
declaration, abstract and well-formed, 177-178
declaration, abstract, 169-170, 176-178
Declarations are Finite, 179, 216
declarations, explicit refinement type, 273, 280
DECL-TYPE', 274
DECL-TYPE, 274
$\overset{\text{def}}{\leq}$, 33
$\overset{\text{def}}{:}$, 59, 64, 210, 212, 214
$\overset{\text{def}}{::}$, 65
$\overset{\text{def}}{\text{empty}}$ (emptyness for refinement type constructors), 184, 210, 215
$\overset{\text{def}}{\text{mtor}}$ Refines, 76
$\overset{\text{def}}{\sqsubset}$, 30, 211
$\overset{\text{def}}{\asymp}$, 47, 206, 215
$\overset{\text{def}}{\leq}$, 210, 212
$\overset{\text{def}}{\sqcup}$, 129
$\overset{\text{def}}{\wedge}$ defined, 198, 210, 216
$\overset{\text{def}}{\wedge}$ Elim, 34, 38, 41, 198, 216
$\overset{\text{def}}{\wedge}$, 33, 210, 212
$\overset{\text{def}}{\wedge}$-defined, 34
Depth of a Recursive Type, 187

depth, 187
deterministic function, 301
`deval`, 264
diagnosing refinement type errors, 305
disj, 8
`disjCnfs`, 9
`dont_memoize`, 288
`double`, 226, 281, 289, 294
$doublepred$, 253

$e \Rightarrow v$ (evaluation), 24
eager value constructors, 183
EC, 114
effective verification of refinement types, 12
efficiency, 6
ELIM-SPLIT, 48-50, 52, 57, 91, 126-127
ELT-SEM, 24, 103
ELT-TYPE, 60, 73, 80, 89, 98, 103, 150, 158
ELT-VALID, 27, 29, 73, 79
Empty Eliminable Assumptions, 188, 190
Empty Intersection, 192, 197
empty refinement type, 3
empty set, 3
empty splits, 62
Empty Transitivity, 200, 202, 214, 220
empty types, 167-169, 183
empty, 2
empty $\stackrel{\text{def}}{=}$ (emptyness for refinement type constructors), 184, 210, 215
Emptyness Consistency I, 189
Emptyness Consistency II, 182, 190, 192, 196
Emptyness Consistency, 215, 220-221
Emptyness Constructor, 184, 220
emptyness for recursive types, algorithmic ($D; S \vdash r$ alg-empty), 186
emptyness for recursive types ($D \vdash r$ empty), 185
emptyness for refinement type constructors ($\stackrel{\text{def}}{=}$empty), 184, 210, 215
emptyness for refinement types ($\vdash r$ empty), 184, 279
Emptyness Subtyping, 185, 193, 220
emptyU, 187
Environment Modification, 81, 85, 140, 153, 159, 164, 235, 260
equality, polymorphic, 7
equivalence classes, 106, 114, 251
equivalence for refinement types ($\equiv$), 36
Equivalence rtort, 212
EQUIV-SPLIT-L, 48, 50, 54-56, 58, 82, 91, 140, 218
EQUIV-SPLIT-R, 48, 50-51, 54, 56-57, 91
erase, 274
error detection, 9
`error`, 7
ev, 240-241
`eval`, 6-7
evaluating boolean expressions, 6
evaluation ($e \Rightarrow v$), 22, 24, 231
expansion, 171
explaining refinement type errors, 305
explicit ML type declarations, 227
explicit ML types, 20, 302
explicit refinement type declarations, 273, 280
expression scheme, 229
expressions, grammar for, 19, 229, 242
expressions, substitution for, 92
expressions, substitution, 23
expressiveness of refinement types, 12
extended recursive type, 166

F (monotone function representing an inference system), 181
`False`, 6
`false`, 7, 223
ff, 223-224
Finite Predefined Refinements, 31, 115, 122, 216
Finite Refinements, 63, 115-116, 164, 238, 252, 274
finite set of refinements of each ML type, 3, 63, 105

Fix Case of Infer is Well-Behaved, 144, 151, 163
fixed point, greatest, 170, 181, 193
fixed point, least, 193
fixed point of monotone function, 168
fixed points, 63, 168, 280-282, 286-287, 298
FIX-SEM, 24, 92, 103, 231
FIX-TYPE, 60, 73, 80, 83, 89, 98, 103, 151, 159, 234
FIX-VALID, 27, 29, 74, 80, 94, 145, 232
flattening, 172
$\triangle$`fn`, 117-118, 121
`fntoref`, 120, 125
Fragments of Principal Split have Useless Splits, 58, 127, 254
fragments, 47
Free Type Variables in Constructors, 232, 239, 243
free type variables, 229
Free Variables Refine, 63-64
function, deterministic, 301
function graphs, minimal, 282
function, monotone, 168, 170

general types, selecting the most, 302
generalized pairs, 247
generalized refinement types, 106, 247
gfp, 182
grammar for expressions, 19, 229, 242
grammar for ML types, 18, 242
grammar for `rectype` statements, 166, 168
grammar for refinement types, 30, 242
grammar for values, 22, 231
grammars, term, 169
graphs, minimal function, 282
greatest fixed point, 170, 181, 193
greatest lower bound, 36
ground boolean expression, 6, 8
ground substitution, 8
ground, 7

higher-order abstract syntax, 230

i (interpretation), 106, 108, 131, 246, 248, 263, 280
I (interpretation), 106, 114, 246, 252
i Gives an Upper Bound, 111, 113, 149, 194, 251
$i(k)(\overline{r}; \overline{r}'')$ Monotone in $\overline{r}$, 249, 252
$i(k)(\overline{r}; \overline{r}'')$ Respects Equivalence in $\overline{r}''$, 249, 252
i Monotone in First Argument, 109, 111, 113, 124, 132-133, 154, 250
i Monotone in Second Argument, 108, 114, 132, 154, 249
I Preserves Equivalence, 114-115, 252
i Preserves Information, 113, 115, 122, 251
`iconstr`, 263
`id`, 223
identifiers, ML type (`mlconid`), 277
identifiers, refinement type (`refconid`), 277
identifiers, type, 277
`ifn`, 117, 120, 124, 131, 264
ignored type arguments, 241, 270
ignored type variables, 13, 258, 279, 305
implementation, prototype, 6, 14, 275, 302, 305
implicit ML types, 19, 302
implicitly declared refinements, 7
indices, de Bruijn, 230
Infer Returns Principal Type, 141, 144, 151, 160, 164, 265, 268
Infer Returns Some Type, 144-145, 163-164, 264
Infer Terminates, 145, 151, 265
`infer`, 135, 144, 238, 263-264, 274, 282
Inferred Types Refine, 19, 27, 59, 68-69, 74, 94, 97, 116, 144, 160, 235
infinite lazy lists, 180
infinite proofs, 181
informative splits, 52
`inst`, 293
instantiation algorithm, 305
instantiation, 7, 14, 63, 224, 242, 281,

288, 295
int, 303
interpretation, abstract, 3, 282-285
interpretation (i), 106, 108, 131, 246, 248, 263, 280
interpretation (I), 106, 114, 246, 252
intersection for recursive types (&), 165-166, 168, 170, 211
intersection for refinement types ($\wedge$), 3, 16, 18, 30, 211
intersection for vectors ($\wedge$), 242
Intersection Membership, 179
intersection, monotonicity of, 36
Intersection Refines, 179, 192, 197
Intersection Value Membership, 183, 209

Join is Decidable, 129
Join, 127
`joinf`, 129

`lastcons`, 1-2
`Later`, 289
lazy lists, 180
lazy representations, 281
lazy value constructors, 183
$lazy$, 181
`lazy_refcon`, 288
least fixed point, 193
least refinement of an ML type, 16
length of a vector, 228
`let` statements, 228, 231, 238-239, 294
LET-SEM', 231
LET-SEM, 231, 238
LET-TYPE, 234, 237-238
LET-VALID, 232
$\perp_{list}$, 2
`List`, 303
$\top_{list}$, 3
list, 1, 7
lists, association, 285
lists, lazy, 180
lists representing substitutions, 7
literal, 8
long, 2
looking up a value in a substitution, 7
`lookup`, 7-8
`looper`, 293
lower bound, greatest, 36

malformed refinement type, 30, 62
`mapsubst`, 293
`maybe`, 46
membership of a value in a recursive type ($D \vdash v \in nr$), 170, 180
memo tables, 305
memoization, 14, 281, 287, 295, 305
Milner-Mycroft type inference, 64
minimal function graphs, 282
Mini-ML, 19
missing case, 6-7
mix, 247, 249, 270
mixed type arguments, 240, 270
mixed type variables, 13, 279, 305
ML Compatibility, 77, 235
ML Free Variables Bound, 29, 64, 232, 243
ML type constructors (`mlconstructor`), 277-278
ML type declarations, explicit, 227
ML type identifiers (`mlconid`), 277
ML type inference, compatibility with refinement type inference, 68
ML type names (`mlconname`), 277
ML type schemes, 223, 228, 279
ML Type Soundness, 27, 29, 99, 232, 243
ML type variables, 7
ML types, explicit, 20, 302
ML types, grammar for, 18, 242
ML types, implicit, 19, 302
ML types, quadruples of ($\overline{\overline{t}}$), 242
ML types, 25, 279
ML typing relation (VM $\vdash e :: t$), 26
ML Value Substitution, 29, 93, 97, 232, 243
`mlconid` (ML type identifiers), 277
`mlconname` (ML type names), 277
`mlconstructor` (ML type constructors), 277-278

`mlscheme`, 279
`mlty`, 279
monomorphic refinement type inference, 13
monomorphic refinement types, 15
monotone function, 168, 170
monotonicity of intersection, 36
mtor Refines, 76-77, 79-80, 235
mtor, 76
multiple refinements of type variables, 224

names, ML type (`mlconname`), 277
names, refinement type (`refconname`), 277
names, type, 277
naming convention, 26
negative type arguments, 240, 270
negative type variables, 13, 258, 279, 286, 305
New Recursive Type Constructors Defined, 178, 187, 216
New Value Constructors Closed, 179
New Value Constructors Defined, 178
New Value Constructors Only, 178
new, 166-167, 176, 178
NEW-INFER-EMPTY, 185-186, 189-193, 200-201, 220
NEW-INFER-RECSUB, 193-194, 196-197, 199-202, 205, 213
NEW-RC-RECVALUE, 180, 183, 191, 205, 208, 221
NEW-RECREFINES, 177, 216
NEW-RECSPLIT, 206-208, 216-218
`nil`, 2
nk, 176
nkc, 176
$nkcs$, 176
nks, 176
$none$, 225
`NONE`, 225
Non-free Variables are Ignored, 63, 82, 95
`not`, 283
`Not`, 6
`notfa`, 290
`now`, 289
np, 176
npc, 176
$npcs$, 176
nps, 176
nr, 176
nrc, 176
$nrcs$, 176
nrs, 176
ns, 16, 106-112, 118-123, 125, 129-134, 141, 144-152, 155, 157-160, 163, 247-248

old, 166-167, 178
OLD-EMPTY, 186, 189-191, 193
OLD-RC-RECVALUE, 180, 191, 205-206, 209, 222
OLD-RECREFINES, 177, 198
OLD-RECSPLIT, 207-209
OLD-RECSUB, 193-194, 198-199, 203, 205
omitting `rectype` statements, 302
Only $\overset{\text{def}}{\text{mtor}}$ Refines, 76
$option$, 225
`Or`, 6
Ordering on i, 112-113, 124, 133, 251

pairs, generalized, 247
pending analysis, 14, 281-282, 284
Piecewise Intersection, 67, 74, 84, 89, 236, 260
polymorphic equality, 7
polymorphic refinement type constructors, 240
polymorphic refinement type inference, 13
polymorphic type constructors, 13
polymorphism, 13
positive type arguments, 240, 270
positive type variables, 13, 258, 279, 305
postponing work during instantiation, 292
practical refinement type inference, 6, 13
$pred$, 246
predefined data types, 303

Predefined Intersection Distributivity Decidable, 271
Predefined Intersection Distributivity Technical, 270
Predefined Intersection Distributivity, 259, 262, 267, 269-270, 297
Predefined Split Intersection, 54, 217
Principal Refinement Types, 115, 238
principal refinement types, 105
Principal Split Existence, 127
Principal Split Implies Useless Splitting Fragments, 58, 254
principal splits, computing, 126
principal splits, 52, 126
principal types, 3, 224, 241
profiling the implementation, 305
progress, syntactic, 80, 88
properly, varies, 258
properties of constructors, 64
prototype implementation, 6, 14, 275, 302, 305

`qallrefs`, 263
QUADRUPLE-REF, 244
quadruples of ML types ($\overline{\overline{t}}$), 242
quadruples of refinement types ($\overline{\overline{r}}$), 242
QUADRUPLE-SUB, 245

$\vdash\ r$ empty (emptyness for refinement types), 184, 279
$\overline{\overline{r}}$ (quadruples of refinement types), 242
$rarrow$, 241, 246, 254
RCON-AND-ELIM-SUB, 34-35, 38, 41, 44, 67, 86, 111, 244-245, 259, 261
RCON-EMPTY, 184, 191
RCON-REF, 31, 38, 72, 122, 244
Rconsimp Sound, 42, 133, 246
`rconsimp`, 42, 133
RCON-SPLIT, 47-49, 54, 90, 253
RCON-SUB Inversion, 45-46, 124, 134
RCON-SUB, 34-35, 38, 42, 44-45, 49, 86, 111-112, 124, 134, 153, 157, 244-245, 257, 261, 268
REC-TUPLE-EMPTY, 186, 190-193
`rectype` statements, grammar for, 166, 168
`rectype` statements, 2, 7, 13, 74, 165, 239, 269, 297, 300-301, 303, 305
recursion on the left hand side of $\rightarrow$, 167, 169, 175, 182
recursion, 63
Recursive Intersection Greatest, 198, 202, 214, 216, 219
Recursive Intersection Lower Bound, 196, 202-203, 218-219
Recursive Split Intersection, 210, 217-218
Recursive Split Soundness, 208
Recursive Subtype Consistency I, 196
Recursive Subtype Consistency II, 196
Recursive Subtype Soundness, 193, 204, 209, 213
Recursive Subtypes Refine, 196, 212, 216-217
recursive type constructors, 165
recursive type, extended, 166
recursive type, membership of a value in ($D \vdash v \in nr$), 170, 180
recursive types, 165
Recursive Unique ML Types, 178-179, 216
`refconid` (refinement type identifiers), 277
`refconname` (refinement type names), 277
references, 305
Refinement and Recursive Split Consistency, 212, 218
Refinement and Recursive Subtyping Equivalence, 212, 221
Refinement Consistency, 178-179
Refinement Constructor Intersection, 41-42, 54, 122, 133, 246
Refinement Constructor Splits are Nonempty, 51, 217
Refinement Constructor Subtyping, 45, 250
Refinement to ML (rtom), 32

refinement type constructors, polymorphic, 240
refinement type constructors, 223
refinement type error, 305
refinement type identifiers (`refconid`), 277
refinement type inference, compatibility with ML type inference, 68
refinement type inference, monomorphic, 13
refinement type inference, polymorphic, 13
refinement type names (`refconname`), 277
refinement type schemes, 223, 228
Refinement Type Soundness, 99, 103-104, 222, 237, 262, 274
Refinement Type Substitution, 237, 262
refinement type variables, 7
refinement types, constant, 281, 286
refinement types, generalized, 106, 247
refinement types, grammar for, 30, 242
refinement types, monomorphic, 15
refinement types, principal, 105
refinement types, quadruples of ($\overline{\overline{r}}$), 242
refinement types, soundness of, 13, 80, 120, 260, 304
refinement typing relation (VR $\vdash e : r$), 58
refines ($\sqsubset$), 65, 177
Refines $\overset{\text{def}}{\leq}$, 34, 38, 216
reflex-$\overset{\text{def}}{\leq}$, 33, 216
REF-TUPLE-EMPTY, 184
regular systems, 169
regular tree automaton, 169, 194, 301
`repeat`, 225
representations, lazy, 281
`reshapeab`, 291, 295
`reshapeba`, 291
reshaping, 290
`rev`, 293
rewrite rules, 225
rewriting `rectype` declarations, 168
rtom, 32
rtort, 211
$rtuple_n$, 242
rules, rewrite, 225
$runit$, 30-31, 34, 167, 178

`samelength`, 174, 301
scheme, expression, 229
schemes, ML type, 223, 228, 279
schemes, refinement type, 223, 228
schemes, type, 64
scoping `rectype` statements, 304
selecting the most general types, 302
Self Recsub, 199, 204, 212, 214, 216
SELF-SPLIT, 48-51, 57, 66, 92, 218, 233
SELF-SUB, 35-36, 42-43, 45, 50, 83-84, 109-111, 125, 152, 199, 233, 245, 266
semantics, 243
separating `datatype` and `rectype` declarations, 167
`setflag`, 276
signatures, 303
simplification, 173
singleton, 2, 301
`sjoinf`, 134
$some$, 225
`SOME`, 225
Soundness of Empty, 191, 205
Soundness of Refinement Type Empty, 185, 191, 221
soundness of refinement types, 13, 80, 120, 260, 304
Split Constructor Consistent, 49, 66, 90, 214, 217
Split Intersection, 48, 54, 58, 82, 208, 210, 254
Split Positive, 253
Split Substitution, 236, 246
Split Subtype Consistent, 49, 216
Split Types Refine I, 51, 56, 70, 254
Split Types Refine II, 51, 71, 254
split, useful, 57
split, useless, 57

Splits are Nonempty, 51, 63, 70, 254
Splits Are Subtypes I, 49, 51, 58, 254
Splits Are Subtypes II, 51, 254
splits, computing principal, 126
splits, informative, 52
Splits of Arrows are Simple, 51, 94, 159, 163-164, 254
splits, principal, 52, 126
SPLIT-SUB, 61-62
splitting for recursive types ($D \vdash nr \asymp s$), 206-207
splitting ($\asymp$), 46, 65, 93, 253, 297, 299, 305
Splitting Value Types, 65, 89, 95, 209, 236, 262
SPLIT-TYPE, 46, 51, 59-63, 70, 80-82, 88-89, 93-95, 99-100, 103-105, 126, 138-139, 141, 144, 146, 236
statements, `case`, 243, 257, 259, 263, 265, 298
statements, `let`, 228, 231, 238-239, 294
statements, `rectype`, 2, 7, 165, 239, 269, 297, 300-301, 303
strengthening, 63
`strictif`, 224
$\top_{string}$, 7
string, 303
substitution for boolean expressions, 7
substitution for expressions, 23, 92
substitutions represented as lists, 7
substitutions, 228-230, 243, 290
Subtype Decidability, 120
Subtype Eliminable Assumptions, 195
Subtype Irrelevancy, 75, 84, 88, 90, 101-103, 105, 221-222, 236, 262
Subtype Strengthening, 194
Subtype Transitivity, 201, 204, 214, 216, 220
`subtypep`, 117, 119, 122, 263, 286
subtypeU(D), 195
Subtypes Refine, 36, 44, 50-51, 56, 63, 70, 122, 158, 196, 212, 236, 246, 251
subtyping for recursive types, algorithmic ($D; S \vdash nr \leq nk$), 195
subtyping for recursive types ($D \vdash nr \leq nk$), 169, 193-194
subtyping for refinement types ($\leq$), 65, 244
sugar, syntactic, 241-243, 249
Syntactic Progress Decidability Sufficient, 138, 151
syntactic progress, 80, 88
syntactic sugar, 241-243, 249
syntax, concrete, 275, 305
syntax, higher-order abstract, 230
`syntp`, 280
systems, regular, 169

$\top_{bool}$, 240-241
$\overline{\overline{t}}$ (quadruples of ML types), 242
tables, memo, 305
$tarrow$, 241, 277
`teqopt`, 280, 286
term grammars, 169
Termination for `subtypep` and `allrefs`, 125
`toCnf`, 9, 305
`TooSmall`, 292-295
toplevel, 171-172
`tp`, 280, 289
trail, 185, 193
trans-$\overset{\text{def}}{\leq}$, 34, 203, 216
Transitivity of $\preceq$, 108, 110, 114, 124, 248
TRANS-SPLIT, 47-48, 50, 53, 56, 58, 91, 127
TRANS-SUB, 35-37, 44-45, 50, 52, 85, 110, 112-113, 121, 123-124, 130-131, 134, 139-140, 153, 155, 158, 164, 245, 267
TRANS-SUBTYPE, 203
tree automaton, regular, 169, 194, 301
`True`, 6
`true`, 7, 223
tt, 15, 223-224, 240-241
$ttuple_n$, 242, 246, 277
$tunit$, 18, 21, 30-31, 178

Tuple Intersection, 40-41, 55, 130, 246
Tuple Refines, 32, 44, 233
Tuple Subtyping, 33, 42, 45, 250
tupling ($\alpha * \beta * \gamma$), 242
TUPLE-AND-ELIM-SUB, 34-35, 39-40, 45, 87, 111, 244
TUPLE-RECREFINES, 177-178, 198
TUPLE-RECSPLIT, 206-207, 209
TUPLE-RECSPLT, 206
TUPLE-RECSUB, 193-194, 198, 200, 203, 206, 222
TUPLE-RECVALUE, 180, 192, 206, 209-210
TUPLE-REF, 31-32, 39, 43, 49, 73, 244
TUPLE-SEM, 24, 102
Tuplesimp Sound, 41, 115, 124, 131, 246
`tuplesimp`, 41-42
TUPLE-SPLIT, 47-49, 55, 90, 253
TUPLE-SUB Inversion, 45-46, 123, 130, 158
TUPLE-SUB, 34-35, 39-40, 42, 44-45, 50, 87, 111, 115, 121, 123, 130-131, 244
TUPLE-TYPE, 47, 60, 73, 79, 87-91, 98, 102-103, 149, 157, 222
TUPLE-VALID, 27-28, 73, 79
type arguments, ignored, 241, 270
type arguments, mixed, 240, 270
type arguments, negative, 240, 270
type arguments, positive, 240, 270
type arguments, 240-241, 270
type constructors, ML (`mlconstructor`), 277-278
type constructors, polymorphic refinement, 240
type constructors, 13, 223
type declarations, explicit refinement, 273, 280
type identifiers, ML (`mlconid`), 277
type identifiers, refinement (`refconid`), 277
type identifiers, 277
type inference, Damas-Milner, 63
type inference, Milner-Mycroft, 64
type names, ML (`mlconname`), 277
type names, refinement (`refconname`), 277
type names, 277
type schemes, ML, 223, 228, 279
type schemes, refinement, 223, 228
type schemes, 64
Type Soundness, ML, 27
Type Substitution Preserves Subtyping, 236, 246, 262, 267
type variable capture, 229-230
type variables, free, 229
type variables, ignored, 13, 258, 279, 305
type variables, mixed, 13, 279, 305
type variables, multiple refinements of, 224
type variables, negative, 13, 258, 279, 286, 305
type variables, positive, 13, 258, 279, 305
type variables, 7, 13, 223-224, 228-229, 239, 258, 279, 286, 305
types, constant refinement, 281, 286
types, empty, 167-169, 183
types, generalized refinement, 106, 247
types, grammar for ML, 242
types, grammar for refinement, 242
types, ML, 279
types, principal, 3, 224, 241
types, quadruples of ML ($\overline{\overline{t}}$), 242
types, quadruples of refinement ($\overline{\overline{r}}$), 242
types, soundness of refinement, 13, 80, 120, 260, 304
typing relation, ML (VM $\vdash e :: t$), 26
typing relation, refinement (VR $\vdash e : r$), 58

U (universe), 181
Union-Find problem, 286
`UNIONFIND`, 286
Unique Inferred ML Types, 27, 70, 116, 145-146, 232, 243
Unique ML Types, 31, 36-37, 65, 72, 74, 160, 179, 233, 244

Unique Predefined Refinements, 31-32, 38, 216
Unique Principal Splits, 52
Unique Refinement, 76, 79, 235
UNIT-REF, 32
universe (U), 181
useful split, 57
useless split, 57

`vallrefs`, 263
Value Arrow Type, 74, 85-86, 235
Value Constructor Type, 74, 86, 235
value constructors, 242
Value Containment, 220
value, membership in a recursive type ($D \vdash v \in nr$), 170, 180
Value Substitution, 29, 51, 92-93, 101, 103-104, 236
Value Tuple Type, 74, 87, 90
values, grammar for, 22, 231
`Var`, 6-7
variable capture, type, 229-230
variables, bound, 230
variables, free type, 229
variables, ignored type, 13, 258, 279, 305
variables, mixed type, 13, 279, 305
variables, multiple refinements of type, 224
variables, negative type, 13, 258, 279, 286, 305
variables, positive type, 13, 258, 279, 305
variables, type, 7, 13, 223-224, 228-229, 239, 258, 279, 286, 305
Variance, 258, 261, 266
Variant Weakening, 258, 261, 266-267
varies properly, 258
VAR-REF, 232, 244
VAR-TYPE, 60, 63, 71, 77, 80, 83, 89, 95, 140, 145, 152, 234-237
VAR-VALID, 27-28, 63, 71, 77, 232
VECTOR-EQUIV, 245
VECTOR-REF, 244
vectors ($\overline{\alpha}$), 228, 242
vectors, intersection for ($\wedge$), 242
VECTOR-SUB, 245
VM $\vdash e :: t$ (ML typing relation), 26
VM, 26
VR $\Vdash e : r$, 81
VR $\vdash e : r$ (refinement typing relation), 58
VR (variable to refinement type mapping), 58
`vsubtypep`, 263

weakened closure of D ($\widehat{D}$), 213
Weakened Intersection Simplification I, 213-214, 221
Weakened Intersection Simplification II, 214, 218
Weakened Intersection Soundness, 213
weakening, 63
WEAKEN-TYPE, 59-61, 64, 67, 70, 75, 79-81, 83-85, 87-89, 91, 94, 99-100, 116, 138-139, 146-150, 206, 209, 221, 236, 257, 259, 261-262, 266, 268
well-formed abstract declaration, 170, 177-178