Space Issues in Compiling with Intersection and Union Types

Allyn Dimock
Harvard University
dimock@das.harvard.edu

Ian Westmacott
Boston University
ianw@bu.edu

Robert Muller
Boston College
muller@cs.bc.edu

Franklyn Turbak
Wellesley College
fturbak@wellesley.edu

J. B. Wells
Heriot-Watt University
jbw@cee.hw.ac.uk

Jeffrey Considine
Boston University
jconsidi@bu.edu

September 2, 2000

Abstract

The CIL compiler for core Standard ML compiles whole programs using the CIL typed intermediate language with flow labels and intersection and union types. Flow labels embed flow information in the types and intersection and union types support precise polyvariant type and flow information, without the use of type-level abstraction or quantification.

Compile-time representations of CIL types and terms are potentially large compared to those for similar types and terms in systems based on quantified types. The listing-based nature of intersection and union types, together with flow label annotations on types, contribute to the size of CIL types. The CIL term representation duplicates portions of the program where intersection types are introduced and union types are eliminated. This duplication makes it easier to represent type information and to introduce multiple representation conventions, but incurs a compile-time space cost.

This paper presents empirical data on the compile-time space costs of using CIL. These costs can be made tractable using standard hash-consing techniques. Surprisingly, the duplicating nature of CIL has acceptable compile-time space performance in practice on the benchmarks and flow analyses that we have investigated. Increasing the precision of flow analysis can significantly reduce compile-time space costs. There is also preliminary evidence that the flow information encoded in CIL's type system can effectively guide data customization.

1 Introduction

Recent research has demonstrated the benefits of compiling with an explicitly typed intermediate language (TIL) [Mor95, PJ96, TMC+96, PJM97, JS98, BKR98, TO98, FKR+99, CJW00, MWCG99, WDMT0X]. One benefit is that explicit types can be used in compiler passes to guide program transformations and select efficient data representations. Another advantage of using a TIL is that the compiler can invoke its type checker after every transformation, greatly reducing the possibility of introducing errors. If strongly typed intermediate languages are used all the way through the compiler to the assembly level (something we do not yet do), the resulting object code is certifiably type safe [Nec97, MWCG99]. Furthermore, types that survive through the back end can be used to support run-time operations such as garbage collection [Tol94] and run-time type dispatch [Mor95].

The benefits of using a TIL are not achieved without costs, which include the space needed to represent the types at compile-time, the time to manipulate the types at compile-time, and the added complications of transforming types along with terms. This report focuses on the compile-time space cost.

Using a naive type representation can incur huge space costs, even if types are only used in the compiler front end for initial type checking. In the worst case, the tree representation of types in Standard ML (SML) programs can have size doubly exponential in the program, and the DAG representation can be exponential in the program size [Mit96]. Although we are mainly concerned with ordinary programs where the worst case space complexity is not experienced, these ordinary programs often have types with impractically large tree representations but acceptable DAG representations. So in practice, DAG representations of types and other techniques are necessary to engineer types of tractable size. For example, the SML/NJ compiler's FLINT intermediate language uses hash-consing, memoization, explicit substitutions, and de Bruijn indices to achieve space-efficient implementation of types [SLM98]. The TIL compiler achieves type sharing by binding all types to type variables, and then performing dead code elimination, hoisting and common subexpression elimination on the types [Tar96, pp. 217–219]. The compiler must then preserve type bindings across transformations, or else repeat the type-sharing transformations. Tarditi reports that the representation size increase imposed by using types in TIL averages 5.15 times without this sharing scheme, but only 1.93 times with sharing.

We have constructed a whole-program compiler for core SML based on a typed intermediate language we call CIL[1] Unlike FLINT and TIL, CIL has three features that make compile-time space issues potentially more challenging to address than in other typed intermediate languages:

1. **Listing-based types:** The CIL type system can encode polyvariant flow analyses using *polyvariant flow types* where labels on type constructors provide flow information and intersection and union types provide polyvariant analysis. Intersection and union types can be viewed as finitary (listing-based) versions of infinitary (schema-based) universal and existential types. For example, CIL uses

   $$\tau_{\mathrm{id}} \equiv \wedge\{f_1 : \mathrm{int} \rightarrow \mathrm{int},\ f_2 : \mathrm{real} \rightarrow \mathrm{real}\}$$

   to represent the SML type $\forall\alpha.\alpha \rightarrow \alpha$ instantiated at types int and real. Encoding polyvariant analyses, which analyze a function multiple times relative to different contexts of use, can introduce components of intersection and union types that differ only by flow information. For instance, when encoding polyvariance, an innocuous type like int $\rightarrow$ int can balloon into something like:

   $$\vee\{g_1 : \mathrm{int} \xrightarrow[\{3,4\}]{\{1\}} \mathrm{int},\ g_2 : \wedge\{h_1 : \mathrm{int} \xrightarrow[\{3\}]{\{2\}} \mathrm{int},\ h_2 : \mathrm{int} \xrightarrow[\{4\}]{\{2\}} \mathrm{int}\}\}.$$

   In the notation $\sigma \xrightarrow[\psi]{\phi} \tau$, the annotation $\genfrac{}{}{0pt}{}{\phi}{\psi}$ is a *flow bundle* in which ϕ (resp. ψ) conservatively approximates the sites in a program that can be sources, or introduction points (resp. sinks, or elimination points) for the values of a flow-annotated type.

   Intersection and union types have several advantages over universal and existential types as a means of expressing polymorphism [WDMT0X]: (1) by making usage contexts apparent, they support flow-based customizations in a type-safe way; (2) finitary polymorphism can type more terms than infinitary polymorphism; and (3) the listing-based nature of finitary types avoids some complications in representing and manipulating quantified types (see section 2.2). There is a space cost for these benefits: the listing-based nature of finitary types, in combination with flow annotations encoding finer grained types, can lead to CIL types are are much larger than those expressed via infinitary types.

2. **Duplicating term representations:** CIL represents the introduction of intersection types by a *virtual record* — a term that explicitly lists multiple copies of the same component term that differ only in their flow type annotations. For example, here is a CIL term that has the type τ_{id} defined above:

   $$\wedge(f_1 = \lambda x^{\mathrm{int}}.x,\ f_2 = \lambda x^{\mathrm{real}}.x).$$

   Similarly, CIL represents the elimination of union types by a *virtual case* expression — a term that explicitly lists multiple type-annotated versions of the same untyped term. Because it makes copies of terms that differ only in type annotations, we call CIL a *duplicating* representation. An advantage of the duplicating approach is that type information for guiding customization decisions is locally accessible in each copy of a duplicated term. An obvious disadvantage of this representation is the duplicated term structure, which is potentially much larger than the more compact introduction and elimination forms used for universal and existential types.

[1] "CIL" is an acronym for "Church Intermediate Language". The authors are members of of the Church Project (http://www.cs.bu.edu/groups/church/) which is investigating the application of intersection and union types in compiling higher-order typed languages.

3. **Closure types exposing free variable types:** CIL does not have universal or existential types because these hide important information about contexts of use and encourage uniform data representations rather than customized ones [WDMT0X]. Assuming whole-program compilation, the finitary polymorphism afforded by flow types is sufficient to compile SML programs. In this respect, the CIL SML compiler is similar to monomorphizing whole-program compilers [TO98, BKR98, CJW00]. However, existential types are particularly useful for abstracting over differences in free variables that are exposed in typed closure representations for functions of the same source type [MMH96, MWCG99, CWM98]. In the CIL compiler, these differences are reconciled by injecting the types of closures into a union type and performing a virtual case dispatch at the application site [DMTW97]. In a type-erasure semantics, these injections do not give rise to any run-time code. However, they can potentially cause a blowup in compile-time space when many functions with different free variables flow together.

   Our approach to closure conversion is similar to that used by TIL-based compilers that remove higher-order functions via defunctionalization [TO98, CJW00]. These maintain type correctness during closure conversion by injecting closures with different free variables that flow to the same application site into a sum-of-product datatype, and performing a case analysis on the constructed value at the application site. As in the CIL compiler, these compilers use flow analysis to customize the closure datatypes for particular application sites. However, these flow analyses are not integrated into the type system, and there is no distinction between virtual closure structures (which exist solely for the purpose of type checking) and real closure structures (which will survive in the run-time code).

Taken together, listing-based types, duplicating term representations, and closure types that expose free variable types raise the specter of compile-time space explosion at both the term and the type level. However, preliminary experiments with a small benchmark suite indicate that standard hash-consing techniques are able to keep the size of CIL types and terms tractable.

The main contribution of this paper are the following two observations:

1. **Duplicating term representations are practical:** Our experiments show that, for the flow analyses that we have investigated, the space required for CIL terms in our benchmarks is always within a factor of two of (and usually significantly closer to) our estimate of a minimal size for a non-duplicating TIL. This result is surprising, since we and many others expected the duplicating term representation to have a significantly higher space cost.

   Before we obtained these results, we expected that it would be essential to develop a *non-duplicating* term representation in which a single term schema somehow contains multiple flow type annotations. For example, using the notation of [Pie91], τ_{id} could be expressed as something like: **for** $\alpha \in \{\mathrm{int}, \mathrm{real}\}.\lambda x^{\alpha}.x$. Although this notation is more compact, it makes type information less accessible and can be tricky to adapt to more complex situations [WDMT0X]. We have made preliminary investigations into other representations, e.g., one based on the skeletons and substitutions of [KW99]. Based on the empirical results presented here, we believe that developing a non-duplicating representation of CIL may be not critical (though it may still be worthwhile). However, it remains to be seen whether these results hold up in the presence of more polyvariant flow analyses.

2. **Finer-grained flow analyses yield smaller types and terms:** Our experiments indicate that increasing the precision of flow analysis can significantly reduce the compile-time space cost for CIL flow types. Benchmarks require the most compile-time space for the least precise type-respecting flow analysis (one that assumes that any function with a given monomorphic type can flow to any call site applying a function with this type). This imprecision leads to union types for closures that are much larger than necessary. More precise flow analyses can substantially reduce the size of these closure types.

   Flow analysis has similarly been used to reduce the size of closure types in monomorphizing and defunctionalizing TIL compilers [TO98, CJW00]. However, previous work has not quantified the benefits of using flow analysis in this context nor studied the effects of different flow analyses on compile-time space.

In addition to our results about the tractability of compile-time space in the CIL compiler, we have preliminary evidence that the compiler may be able to achieve one of its main design goals: avoiding *representation pollution* when choosing customized data representations. Representation pollution occurs when a source form is constrained to have an inefficient representation because it shares a sink with other source forms using the inefficient represention. A complementary phenomenon occurs with pollution of sink representations.

As an example of representation pollution, as well as some other issues that arise in a compiler based on CIL, consider compiling the SML-like source term:

$$\begin{array}{l}\textbf{let } f^{\text{int}\to\text{int}} = (\lambda x^{\text{int}}.x * 2)\\ \textbf{in let } g^{\text{int}\to\text{int}} = (\lambda y^{\text{int}}.y + a^{\text{int}})\\ \quad \textbf{in } \times\big(f \mathbin{@} 5, (\textbf{if } b^{\text{bool}} \textbf{ then } f \textbf{ else } g) \mathbin{@} 7\big)\end{array}$$

In addition to several other forms, the above term contains two abstractions and two applications (denoted by the @ symbol). The abstraction $(\lambda x^{\text{int}}.x * 2)$ flows to both application sites while the abstraction $(\lambda y^{\text{int}}.y + a^{\text{int}})$ flows only to the rightmost application site.

The diagram in figure 1 gives an abstract depiction of a CIL compiler intermediate representation of the above term that might emerge from the Type Inference / Flow Analysis (TI/FA) stage of the compiler.

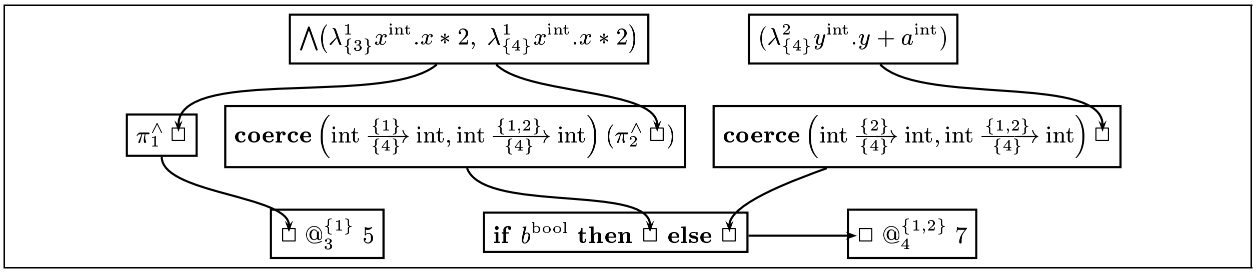

Figure 1: A possible result of Type Inference / Flow Analysis.

The TI/FA stage (described in more detail in section 2.3) computes an approximation of the flow of values between sources and sinks in the input term and represents the analysis in the output typing. The CIL representation of the source term $(\lambda x^{\text{int}}.x * 2)$ is a virtual tuple[2] of the form:

$$\bigwedge\Big(\lambda^1_{\{3\}} x^{\text{int}}.x * 2,\ \lambda^1_{\{4\}} x^{\text{int}}.x * 2\Big)$$

which contains one copy of the function for each of its consumption sites. The terms of the form $(\pi^{\wedge}_i \square)$ are *virtual tuple projections* which select the ith component of a virtual tuple.

Although the duplicate components of a virtual tuple consume space at compile-time, they will share the same run-time representation, and no space needs to be allocated for the virtual tuple at run-time. If the compiler elects to customize the representations of the components of a virtual tuple, the virtual tuple will be *reified* into a real tuple that is explicitly represented in the run-time code. The compiler is designed so that reifying virtual forms is type-safe.

The type of the first component of the virtual tuple is the type required for the function position of the application site to which the function flows. The type on the second component of the virtual tuple does not match that required at its application site so this component must be coerced to the correct type somewhere along the flow path to the application site.

As representation decisions are made during subsequent stages of compilation, further duplication may occur. Figure 2 depicts a possible output of the Flow Separation stage. This stage (described in more detail in section 2.3) may introduce virtual forms wherever a function type needs to be transformed into multiple representation types. In figure 2, the Flow Separation stage has split the application site $(\square \mathbin{@}^{\{1,2\}}_4 7)$ into two applications sites $(h \mathbin{@}^{\{1\}}_4 7)$ and $(h \mathbin{@}^{\{2\}}_4 7)$. These applications occur within a virtual case expression. The functions formerly flowing to the single application site are now injected into a union type. These virtual variants both flow to the discriminant position of the virtual case expression. The virtual case dispatches on the type of the discriminant to one of the two duplicate applications.

[2] A virtual tuple can be considered a virtual record whose field names are integers.

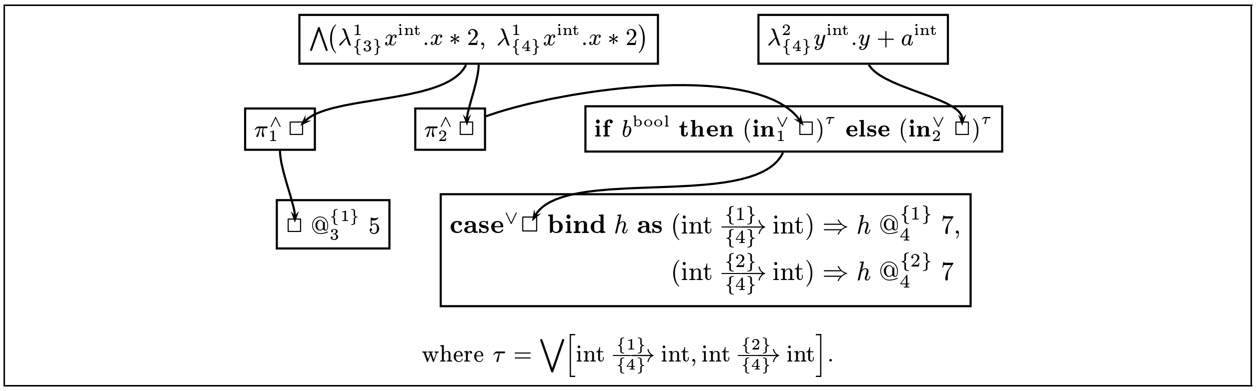

Figure 2: A possible result of Flow Separation.

As with source splits, this kind of sink duplication has important consequences for the amount of space consumed at compile-time. If the virtual variants and virtual case expressions are reified in a subsequent stage then this duplication will also have important consequences for the amount of space required for the emitted object program. Observe that the sink duplication introduced by Flow Separation has eliminated the need for both of the coercions present in figure 1 and will usually reduce the sizes of flow sets. In general, there are many trade-offs between the amount of virtual duplication and subtype coercion. The trade-offs are very sensitive to the granularity of the flow analysis and to the representation customization strategy.

Our preliminary analysis indicates that a large percentage of virtual forms are reified. This suggests that the compiler is being effective at reducing representation pollution. However, this may be an artifact of the "oversplitting" behavior of our current reification algorithm. More work is necessary to evaluate the customization capabilities of the CIL SML compiler. In a future report we expect to present a detailed study of the run-time consequences of compiling with polyvariant flow types.

The remainder of this paper is organized as follows. Section 2 provides an overview of the CIL compiler for SML, section 3 presents space-related measurements for several standard benchmark programs at various phases of compilation, and section 4 summarizes our conclusions and describes future work.

2 An Overview of the CIL Compiler

2.1 The Intermediate Language

To implement the features of core SML, CIL, extends the purely functional λ^{CIL}-calculus [WDMT0X] with primitive datatypes, references, arrays, and exceptions. The syntax and typing rules of CIL are summarized in appendix A. Although CIL is based on the λ^{CIL}-calculus, CIL itself is not a calculus. We have implemented a semantics for CIL, but we have not written its formal counterpart. While we have proven formal properties like standardization, subject reduction, and type soundness for λ^{CIL}-calculus, we have not yet established any of these properties for CIL.

2.2 Type and Term Representations

To keep the space used storing types to a reasonable level, the CIL compiler uses hash-consing to represent types as compact directed acyclic graphs instead of as trees. This is similar to the type representation in the SML/NJ compiler's implementation of its FLINT intermediate language [SLM98]. One important issue faced in FLINT is not an issue for CIL. FLINT types have higher-order features such as abstractions and applications, i.e., a λ-calculus inside the types. Because FLINT types are identified modulo β-conversion, and because eager β-normalization of types can lose sharing and do excess work, the hash-consing scheme for FLINT types uses explicit substitutions [KR95] and a fancy memoization of substitution propagation steps. Unlike FLINT, the CIL types do not have such higher-order features, so the CIL hash-consing of types is simpler.

Sets of flow labels are often used by many types and/or terms. A single copy of each set is shared by all uses. Using the duplicating representation for terms, two CIL term occurrences are rarely structurally equivalent, so we do not use hash-consing for terms. However, the types and flow sets annotating terms are hash-consed, as described above.

2.3 Compiler Architecture

The architecture of the CIL compiler, which has been presented previously [DMTW97], is summarized in Figure 3. This section briefly describes the compilation stages depicted in the figure.

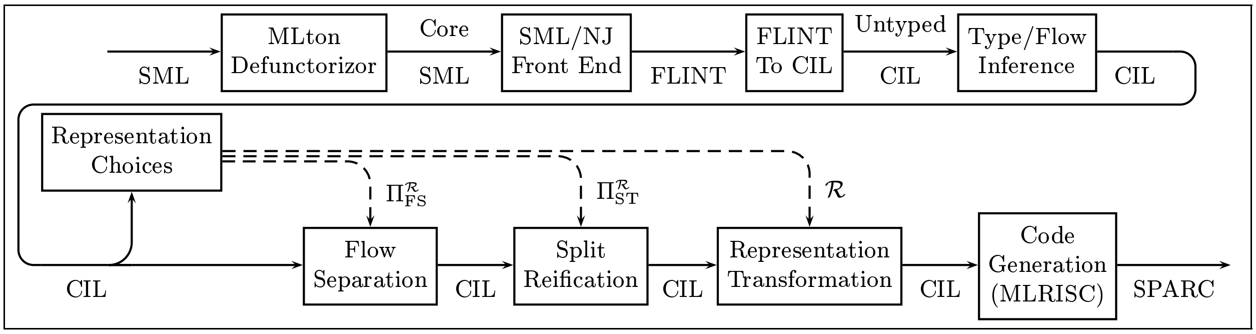

Figure 3: Compiler Architecture.

Front End

In implementing the compiler, we took advantage of existing tools and other freely available SML compilers. The CIL compiler uses the MLton source-to-source defunctorizesr [CJW00] as a prepass to convert SML into Core SML. It then uses the front end of the SML/NJ 110.03 compiler (somewhat modified) to produce FLINT code. The FLINT code is translated to untyped CIL code, keeping datatype information on the side to avoid reinference of recursive types.

Type Inference / Flow Analysis

The TI/FA stage accepts an untyped CIL term (plus some of the FLINT type information) as input and returns a typed CIL term as output. The typed term encodes a flow analysis that is a conservative approximation of the run-time flow. The TI/FA module supports flow analyses that vary with respect to the precision of the approximation.

To date, we have implemented six different flow analyses. In this paper, we present data from two of these: what we call *typed source split* and *min type respecting*. The *typed source split* analysis is an variant of Banerjee's [Ban97] which gives precision similar to the combination of monomorphization and "simple" flow analysis (a la [Hen92]) used in [TO98]. It introduces virtual tuples and virtual projections but neither virtual variants nor virtual case forms.

The *min type respecting* analysis is the least precise flow analysis that is still type correct (cf. [JWW97]). It conflates the flow information on all values of the same flow erased type. For example, an abstraction of type **int** $\rightarrow$ **int** will be assumed to flow to every application site whose rator has this type. This analysis models a monomorphizing compiler in which types carry no useful flow information.

We have also implemented a finer analysis that splits on some variable occurrences. The other analyses range in precision between the *typed source split* and *min type respecting*.

Representation Choices

The representation choices stage selects representations for a function that are adequate for each of the application sites to which it flows. Four different function representation choice strategies have been implemented.

The *uniform* strategy represents all functions with closure records of the type

$$\times\{\text{code} : \{\text{arg} : \tau_{\text{arg}},\ \text{env} : \tau_{\text{env}}\} \to \tau_{\text{body}}, \text{env} : \tau_{\text{env}}\},$$

where the *code* field contains a closed one-argument function and the *env* field contains a record of the values of the free variables of the function. A closure data structure is applied to an argument by projecting both fields from the closure record and applying the function from the code field to an argument record consisting of (the closure conversion of) the actual argument packaged together with the projected environment.

The other three representation strategies generate specialized representations based on various conditions detected in the term structure. Steckler and Wand [WS94] coined the term "selective" representation to refer to representations of functions that do not include an environment component. A selective representation is adequate for a closed function if the function flows only to call sites with compatible application protocols. In [WS94], selective representations were disabled in the presence of representation pollution — i.e., when a closed function shared a call site with some number of open functions.

The *selective sink splitting* strategy implemented in the CIL compiler generates a selective representation when the function has no free variables. This representation is called "sink splitting" because if the function shares call sites with open functions, the transformation framework will inject the function representations into a sum type and the application site will be split into multiple sites governed by a case dispatch.

The *selective source splitting* strategy generates a selective representation for a closed function flowing to call sites that are not shared with open functions. Under this strategy, if a closed function shares some application sites with other closed functions but shares other application sites with open functions, then the framework will "split the source" by generating a record containing several copies of the function. The appropriate representations are projected from the record somewhere along the flow path to the respective call sites.

The final representation strategy inlines (possibly open) functions at the call site. The inlined representation of a function consists of the record containing the values of the function's free variables. It is possible to specify many different inlining heuristics. Currently, the inliner will select an inlined representation for any non-recursive function flowing to two or fewer call sites.

Flow Separation

The Flow Separation (FS) stage accepts as input a typed program and a flow-path-partitioning function. The latter is supplied by the representation choices stage. It specifies which flow paths can coexist in the same flow bundles. For flow paths that cannot coexist in the same bundle, the Flow Separation phase will introduce whatever coercions and virtual forms (i.e., union injections, case-unions, virtual tuples or virtual tuple projections) are required to ensure that the result of the transformation will be well-typed. The Flow Separation algorithm is specified in [DMTW97].

Split Reification

The Split Reification (SR) phase accepts as input a typed term and a flow-path-partitioning function. This phase reifies whatever virtual forms are required to ensure that the result of the transformation is well-typed. We refer to the reification process as *splitting* as it causes the code generator to generate multiple copies of a term, where without reification, only one copy would have been produced. In general, the current simple algorithm may split more than is necessary. Specifying and implementing a more efficient splitting algorithm remains for future work.

Representation Transformation

The Representation Transformation (RT) stage accepts as input a typed term and a representation map provided by the representation choices stage. It walks the term and installs the function representations specified by the map. One interesting aspect of the transformation is that the result of the transformation may have a recursive type even though the source of the transformation has no recursion in either terms or types.

Code Generation

The CIL compiler back end transforms typed CIL programs into assembly code for the SPARC processor. It does not currently add any type annotations, or assertions, to the assembly code. The produced assembly code is linked with a runtime library providing the environment in which CIL programs are executed. The back end is based on MLRISC, a framework for building portable optimizing code generators [Geo97]. CIL programs are translated into the MLRISC intermediate language, and the framework is specialized with CIL conventions for each target architecture.[3] MLRISC handles language-independent issues such as register allocation and code emission.

The runtime library is written in C and provides memory management, exception handling, basis functions and a foreign function interface for CIL programs at runtime. The runtime library currently manages memory using the Boehm-Demers conservative garbage collector for C [Boe93]. CIL programs use stack-allocated activation records, which have a layout similar to C stack frames. Basis functions are called through the foreign function interface, which provides data and activation record conversions between CIL and foreign languages.

CIL data representations are straightforward. Records, arrays, references, and strings are heap-allocated and include size headers[4]. Exception identifiers and all other constants are immediate. Injections may either be immediate or heap allocated, depending on the number and type of summands in their type.

Recursive bindings are restricted to values, as defined in figure 6 (see Appendix A). The extended notion of value presented there ensures that terms bound to variables in recursive definitions can not diverge, affect the store, or raise exceptions. Although input programs must adhere to SML restrictions on recursive definitions (because we use the SML/NJ elaborator), compiler transformations may (and do) create recursive definitions which bind extended values to variables. The extended value restriction allows the code generator to use a two phase algorithm for recursive bindings: The first phase allocates memory for the values, while the second phase fills them in. The code generator does not yet optimize tail recursion.

3 Representation Measurements

Our interest in this paper is to determine whether CIL has acceptable compile-time space costs and to evaluate how flow analysis and representation strategy combinations affect these costs. In this section we present data indicating that CIL is tractable as a compiler intermediate language when used in conjunction with a reasonably fine-grained flow analysis.

3.1 Space Profiles

We have tested the CIL SML compiler for most combinations of flow analyses and function representation strategies on 23 kernels and small benchmarks taken from the O'Caml, TIL and SML/NJ benchmark suites. In figures 4 and 5, we present space profiles for five representative benchmarks for two flow analyses and two function representation strategies. We show data for the *uniform* function representation strategy to indicate the amount of data needed to correctly closure convert functions without customizing representations. We show the *selective sink splitting* strategy as an example of a strategy that customizes function representations. The *typed source splitting* flow analysis is currently our most accurate analysis that does not split on variable occurrences. The *min type respecting* flow analysis is included to show size bloat that can occur when flow analysis provides no information beyond the type.

Each space profile shows intermediate representation size information at various CIL compiler stages. The legend in Figure 4 explains how to interpret the data. Of particular importance is the position of the horizontal tick mark found in the portion of the white bar representing the term size. The portion of the white bar below the tick mark is our conservative estimate of the space that might be required for a non-duplicating representation of the term. The position of the horizontal tick mark is computed as the term size ignoring all but the leftmost branches of virtual records and virtual case expressions. Virtual record

[3] Although an advantage of the MLRISC framework is its portability, it still requires substantial work to port a code generator based on MLRISC. For this reason we have concentrated only on the SPARC architecture to date.

[4] Such headers are currently unnecessary since we use conservative GC. But it is expected that in the future we will develop customized memory management.

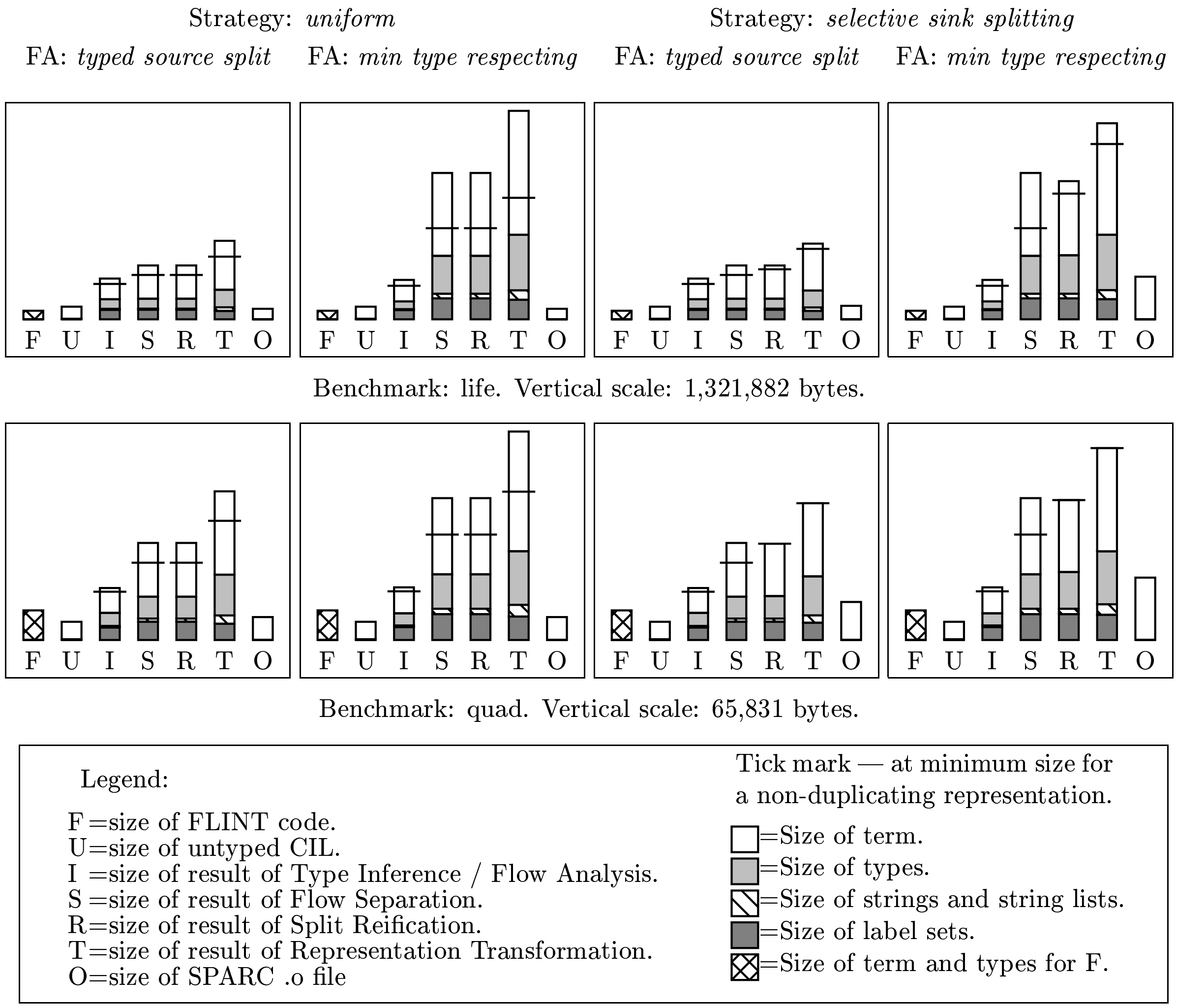

Figure 4: Sizes of benchmark phases by strategy and flow analysis I

nodes and virtual case nodes are included in the count because they serve as markers for intersection type introduction and union type elimination points. We assume that such markers would be required in any non-duplicating representation. Virtual projection and virtual injection nodes and included to approximate (resp.) the markers required for intersection type elimination and union type introduction forms. Finally, the count also includes coercion nodes.[5]

The size information was gathered by adding a function to the SML/NJ runtime system which runs the *mark* stage of the SML/NJ garbage collector using a particular object as the root. The function reports the size of all marked objects that are reachable from the root object. We present all size information in bytes rather than in type or term constructor nodes. We find that the average size of our type nodes and of our term nodes for a given benchmark is generally in the range of 10 to 12 times the size of a machine word.

[5] An even more conservative approximation of the space required for a nonduplicating representation would be the size of the type-erased term. We believe that this is unrealisitically small.

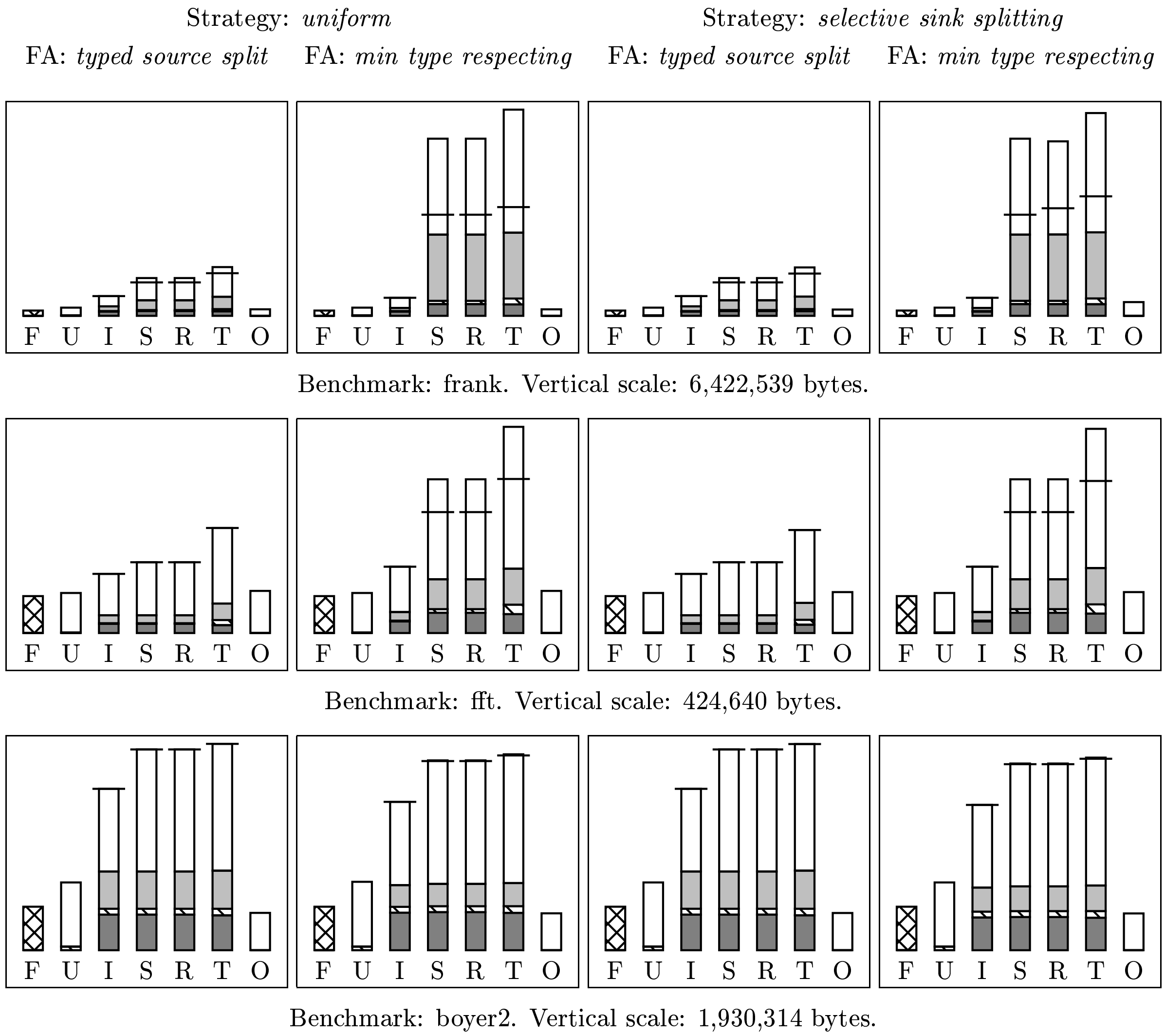

Figure 5: Sizes of benchmark phases by strategy and flow analysis II

3.2 Interpretation of the Space Profiles

Interpreting the size of the untyped term

In most cases the untyped CIL code, **U**, is slightly larger than the typed FLINT code, **F**. This is due in part to the fact that the CIL representation carries more information about records and datatypes than does the FLINT representation. Of the benchmarks and kernels that we show, **quad** takes less space for untyped CIL than for FLINT ; in all other cases the untyped CIL code is larger than the FLINT code.

The **F** and **U** columns are not quite comparable for several reasons. The **F** column overestimates the size of the FLINT code in the sense that it includes the size of FLINT type information. FLINT and CIL also differ in terms of which basis functions are compiled with the program and which are pre-compiled in the run-time system.

Columns **F** and **U** are independent of the flow analysis or the function representation strategy, but appear in multiple graphs as reference points.

Interpreting the output of the Type Inference / Flow Analysis Stage

Column **I** shows the size of the typed and flowed term output from the TI/FA stage. As illustrated by the representative space profiles, the TI/FA pass can expand the size of the term by introducing virtual nodes.

In monomorphic benchmarks, (e.g., **boyer2**, **fft**, and **frank**), term size is only increased by the addition of **coerce** forms that indicate where subtyping is used. In benchmarks with polymorphic functions (e.g., **life**, and **quad**), the TI/FA stage makes one virtual copy (using $\wedge$) of each polymorphic function at each flow-erased type at which the function is used.

In the two flow analyses shown, the distance of the tick mark from the top of the **I** bar reflects the amount of type polymorphism in the benchmark. In general, the tick mark indicates the amount of *polyvariance* of the analysis, which, for some analyses, may be substantial even for monomorphic code.

Interpreting the output of the Flow Separation Stage

Column **S** shows the size of the output from the FS stage. The FS stage introduces whatever new virtual constructs are required to ensure that the result of the (later) RT stage will be well-typed. For example, abstractions that share a call site may have the same type, up to flow information, after the TI/FA stage, but may differ from each other in the number, name and types of free variables. The FS stage must create types that differ in structure as well as in flow information for these different terms.

Under the uniform strategy, the growth in size from **I** to **S** is due only to differences in the environment component of closures – differences that will not be reflected in the object code. In other strategies, some of the growth may be due to function representations that require different object code.

The growth in size from **I** to **S** depends on the accuracy of the flow analysis. In the *min type respecting* flow analysis, the labels for all abstractions of a given (flow erased) type appear in the source label set for each application site for that type. This requires the flow separator to introduce larger intersection and union types, and to perform more virtual term duplication than would be required for a finer flow analysis. This is seen consistently throughout the data, with **frank** being the most dramatic example, and **boyer2** being the least dramatic. The growth in **frank** is due to a large number of curried (and higher-order) functions – implying open functions requiring separation due to differences in free variables. In **boyer2**, all abstractions are closed up to names of known functions[6], so there are few free variables requiring separation. Most abstractions in **boyer2** are first order, so the number of flow-erased types that the flow separator needs to convert is much smaller than other programs.

Interpreting the output of the Split Reification Stage

Column **R** shows the size of the output from the SR stage, which reifies some virtual constructs — splitting them to take advantage of differing representations. The number of term and type nodes remains the same because the transformation is merely changing virtual entities to real ones.[7]

Under the *uniform* strategy, the **S** and **R** columns show identical tick mark positions. This is expected because we implement only a single function calling convention for the uniform strategy. Under the *selective sink splitting* strategy, the position of the tick mark may change upwards due to reification of virtual constructions: this is what we expect from splittings introduced to circumvent representation pollution and to insert customized data representations. This is shown most dramatically in **quad** (a kernel repeatedly applying a doubling function), in which all virtual constructs are reified. In contrast, the **fft** benchmark shows no pollution of function representations when compiled with the *selective sink splitting* strategy.

If we see even a little reification for a strategy, we know that some part of the transformed program will use a simpler representation. If this change is in an inner loop, then a single reification may dramatically affect program performance. To determine the effectiveness of a strategy, we need to show data about the performance of the transformed programs — which is outside the scope of this paper.

Our current SR stage is quite simple: If it encounters two different representations in a single virtual construct, then it converts the virtual construct into the equivalent real construct. Our current splitting algorithm can oversplit because it reifies a virtual form whenever it contains components that require different representations. But given an n-way virtual form whose components require $m < n$ different representations, the virtual form could be replaced with a real form containing m virtual forms. Oversplitting will result in unnecessary duplicated code in the object file. Oversplitting impacts the performance of the generated

[6] In the current version of the CIL compiler, known function names are treated as free variables. This will improve in future versions.

[7] The size of the term component decreases slightly in some profiles due to assymetries between virtual and real injections in the current implementation (e.g., **life**, with strategy = *selective sink splitting* and flow analysis = *min type respecting*).

code when the m-way real form could be more efficiently compiled than the n-way form. We have not yet measured the amount of oversplitting arising from the current algorithm nor have we experimented with other splitting algorithms.

Output of the Representation Transformation Stage

The type information in a closure converted term is larger than in the pre-converted term. This is visible in the profiles for all the benchmarks. Part of this growth is in the creation of types for the required closure and argument records. Part of this growth is the creation of types for environments. In our framework, programs with more open terms will experience more growth in types.

The introduction of closure and argument records and the passing of variables in environments causes an increase in term size. In our implementation of closure conversion, the major increase in term size is from projections from the environment: our implementation puts in a projection from the environment wherever a free variable occurs. The creation and destructuring of closure and argument records will show different percentage effects in different benchmarks depending on the relation of the number of abstractions and applications to other term constructors.

The **boyer2** benchmark has the highest ratio of closed to open terms, so its term size grows, essentially, only by introduction of closure and argument records. In this case the growth in size is relatively small. In contrast, **fft** has a high percentage growth.

The ratio of the size of the CIL representation to the size of a non-duplicating TIL can decrease in the RT stage, but can never increase since types size can only grow, and since no more duplication is introduced into the term at this stage.

Duplicating vs. nonduplicating intermediate representations

Columns **I**, **S**, **R** and **T** have tick marks showing a lower bound on the size of a typed and flowed term in a non-duplicating TIL. The position of the tick mark shows that in the benchmark programs presented (and so far in all benchmarks that we have tried), the space used in CIL's duplicating term representation is never more than twice our estimate for a non-duplicating representation. This is both surprising and encouraging. However, it remains to be seen whether these results hold up in the presence of more polyvariant flow analyses.

Coarse vs. fine flow analysis

We have shown that the choice of flow analysis can greatly influence the growth in term size needed to produce well-typed function representations. The most dramatic example occurring in the benchmark **frank**, where, for the *uniform* function representation strategy the *min type respecting* analysis resulted in a size after Flow Separation 4.7 times the size of that produced using the *typed source split* analysis.

We have accumulated some data so far for a flow analysis using only equality constraints. This analysis is intermediate in precision between *typed source split* and *min type respecting*, and more closely resembles the former. As expected, profiles generated using this analysis are intermediate between those for *typed source split* than to the profiles for *min type respecting*, and quite close to the profiles for *typed source split*.

The cost of accurate closure types

The profiles give us some idea as to the compile space cost of accurately representing closure types. With *uniform* function representation and *typed source split* analysis the growth in size from the output of Type Inference / Flow Analysis stage to the output of the Representation Transformation stage shows the space needed for closure types and for virtual cases where multiple closures flow together. This growth ranges from the size of RT output 1.28 times the size of TI/FA output for **boyer2** to 2.86 times for **quad**. The ratio of the types sizes is 1.02 for **boyer2** and 3.13 for **quad**. **quad** is atypical, being a very small program constructed to have relativly large types.

4 Conclusions and Future Work

We have shown that the amount of space used in compiling SML with CIL terms and types is practical on our benchmarks for the more precise flow analyses that we have investigated. Most importantly, the term sizes in our straightforward duplicating representation are never more than twice our underestimate of term sizes using in a non-duplicating representation. Transformations that use type and flow information on virtual terms to generate customized data representations would be more difficult to engineer in a non-duplicating representation. A factor of less than two in space is acceptable to avoid further complicating the transformations.

The typical non-trivial growth in size from the result of TI/FA to the result of RT is obviously undesirable, and might be smaller in an intermediate representation that could hide environment types with an existential quantifier. This raises the question of whether the more precise type information calculated in the CIL without $\exists$ is useful in terms of transforming a program for better run-time performance. If not, we should extend CIL with existential types.

Although the standard hash-consing technique sketched earlier is the one used to generate the statistics for this paper, we have almost finished changing to a new hash-consing scheme, which we expect to give much better performance. The motivation for the new scheme is due to the combination of (1) the pervasive use of recursive types in CIL and (2) the fact that the CIL type system identifies recursive types with the infinite trees that result from unwinding them infinitely. The new scheme represents types as directed graphs and implements recursion using cycles. This will avoid any lack of sharing of different α-equivalent representations of recursive types by simply making the variable names go away completely. It will also avoid the need to have type manipulation special-case the recursion form (which can currently appear anywhere). The new scheme uses a method of incremental DFA minimization to maintain the invariant that each possible type is represented by at most one node in the graph. This will allow constant-time type equality checking, which our current hash-consing scheme does not support due to the possibility of differing representations of the same recursive type.

Our new method of incremental DFA minimization to represent all types in the same graph is similar to a method suggested by Mauborgne [Mau00], but was developed completely independently. Our method needs $O(n \log n)$ space to store the types, while Mauborgne's needs $O(n^2 \log n)$ space, where n is the number of distinct types and some upper-bound on the arity of type constructors is assumed. Also, even in cases where Mauborgne's method approaches linear space complexity, ours will typically use half as much space.

Encoding more flow analyses in CIL remains an important area for future work. Recent work has shown that many standard flow analyses, such as k-CFA [Shi91, JW95, NN97] and the cartesian product argument-based analysis [Age95] can be encoded into a type system with intersection and union types and flow labels [PP99, AT00]. However, unlike CIL, these type systems have deep subtyping. We are exploring a translation between deep and shallow subtyping that will allow us to employ these recent theoretical results in the CIL compiler. We are eager to see how highly polyvariant flow analyses affect our results regarding the duplicating term representation.

There are many areas for improvement in the CIL compiler as a whole. In particular we have developed the framework for generating customized data representations, but work remains to be done in optimizing those representations. Several existing algorithms can be more efficiently implemented, such as the splitting algorithm. The compiler can benefit from many standard optimizations not yet implemented, as well as non-standard ones, such as the complete removal of polymorphic equality. There are also opportunities for improvement in the representation of the intermediate language. For example, we store record labels in the terms without any sharing.

Finally, this report has focused only on compile-time space issues. In the future, we expect to report on compile-time time complexity as well as run-time space- and time-complexity.

References

[Age95] O. Agesen. The Cartesian product algorithm. In *Proceedings of ECOOP'95, Seventh European Conference on Object-Oriented Programming*, vol. 952, pp. 2–26. Springer-Verlag, 1995.

[AT00] T. Amtoft and F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In ESOP '00 [ESOP00], pp. 26–40.

[Ban97] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In ICFP '97 [ICFP97].

[BKR98] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In ICFP '98 [ICFP98].

[Boe93] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. ACM SIGPLAN '93 Conf. Prog. Lang. Design & Impl.*, pp. 197–206, 1993.

[CJW00] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In ESOP '00 [ESOP00], pp. 56–71.

[CWM98] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. In ICFP '98 [ICFP98], pp. 301–312.

[DMTW97] A. Dimock, R. Muller, F. Turbak, and J. B. Wells. Strongly typed flow-directed representation transformations. In ICFP '97 [ICFP97], pp. 11–24.

[ESOP00] *Proc. European Symp. on Programming*, vol. 1782 of *LNCS*. Springer-Verlag, 2000.

[FKR+99] R. Fitzgerald, T. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. Technical Report 99-33, Microsoft Research, 1999.

[Geo97] L. George. MLRISC: Customizable and reusable code generators. Technical report, Bell Labs, 1997.

[Hen92] F. Henglein. Simple closure analysis. Technical Report D-193, DIKU, Mar. 1992.

[ICFP97] *Proc. 1997 Int'l Conf. Functional Programming.* ACM Press, 1997.

[ICFP98] *Proc. 1998 Int'l Conf. Functional Programming.* ACM Press, 1998.

[JS98] S. L. P. Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, Sept. 1998.

[JW95] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Conf. Rec. 22nd Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 393–407, 1995.

[JWW97] S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages. In *Proc. 4th Int'l Static Analysis Symp.*, vol. 1302 of *LNCS*. Springer-Verlag, 1997.

[KR95] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitution. In *7th Int'l Symp. Prog. Lang.: Implem., Logics & Programs, PLILP '95*, vol. 982 of *LNCS*, pp. 45–62. Springer-Verlag, 1995.

[KW99] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pp. 161–174, 1999.

[Mau00] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In ESOP '00 [ESOP00], pp. 275–289.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[MMH96] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Conf. Rec. POPL '96: 23rd ACM Symp. Princ. of Prog. Langs.*, 1996.

[Mor95] G. Morrisett. *Compiling with Types.* Ph.D. thesis, Carnegie Mellon University, 1995.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. on Prog. Langs. and Systs.*, 21(3):528–569, may 1999.

[Nec97] G. C. Necula. Proof-carrying code. In POPL '97 [POPL97], pp. 106–119.

[NN97] F. Nielson and H. R. Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In POPL '97 [POPL97], pp. 332–345.

[Pie91] B. C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.

[PJ96] S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. European Symp. on Programming*, 1996.

[PJM97] S. L. Peyton Jones and E. Meijer. Henk: A typed intermediate language. In *Proc. First Int'l Workshop on Types in Compilation*, June 1997.

[POPL97] *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, 1997.

[PP98] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, pp. 197–208, 1998. Superseded by [PP99].

[PP99] J. Palsberg and C. Pavlopoulou. From polyvariant flow information to intersection and union types. A substantially revised version of [PP98]. Available at http://www.cs.purdue.edu/homes/palsberg/paper/popl98.ps.gz, Feb. 1999.

[Shi91] O. Shivers. *Control Flow Analysis of Higher Order Languages.* Ph.D. thesis, Carnegie Mellon University, 1991.

[SLM98] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In ICFP '98 [ICFP98], pp. 313–323.

[Tar96] D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* Ph.D. thesis, Carnegie Mellon University, Dec. 1996.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. Prog. Lang. Design & Impl.*, 1996.

[TO98] A. P. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Funct. Prog.*, 8(4):367–412, 1998.

[Tol94] A. Tolmach. Tag-free garbage collection using explict type parameters. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, pp. 1–11, 1994.

[WDMT97] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, pp. 757–771, 1997. Superseded by [WDMT0X].

[WDMT0X] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Prog.*, 200X. To appear. Supersedes [WDMT97].

[WS94] M. Wand and P. Steckler. Selective and lightweight closure conversion. In *Conf. Rec. 21st Ann. ACM Symp. Princ. of Prog. Langs.*, pp. 435–445, 1994.

A The Intermediate Language

Before presenting the language, it is important to briefly explain some notation. In figures 6 and 7, the notation $(P(i))_{i=j}^{k}$, where P(i) is some expression mentioning i, stands for $(P(j), P(j+1), \ldots, P(k-1), P(k))$, where $P(x)$ means the result of replacing the symbol i in $P(i)$ by x. The notations $\{P(i)\}_{i=j}^{k}$ and $[P(i)]_{i=j}^{k}$ are the same except for using different delimiters. Similarly, the notation $\forall_{i=j}^{k}.\ P(i)$, where $P(i)$ is some proposition mentioning i, stands for $(\text{true} \wedge P(j) \wedge P(j+1) \wedge \cdots \wedge P(k-1) \wedge P(k))$.

Figure 6 presents the syntax of CIL (Church Intermediate Language). There are two levels of language, untyped and typed, that are related by a notion of type erasure. The untyped language is necessary for specifying the legality of virtual tuple forms ($\wedge(\ldots)$) and virtual case expression forms ($\mathbf{case}^{\vee}$), and also for specifying the semantics of the language (not defined here). The syntax begins by defining contexts rather than terms to avoid duplicating the definitions to obtain contexts. Contexts are needed to define *parallel* contexts and subterm occurrences (not defined in this paper), notions necessary in defining the semantics of the language as well as algorithms in the compiler. Untyped terms are defined as holeless untyped contexts, while type-annotated terms are type-annotated contexts whose type erasure is an untyped term.

The untyped contexts include constants (c), variables (x), primitive applications ($pr(\ldots)$), abstractions (λ), applications (@), binding forms (**let**), recursive binding forms (**letrec**), record introduction ($\times$), and elimination ($\pi^{\times}$) forms, variant introduction (ι^{+}), and elimination ($\mathbf{case}^{+}$) forms, exception tag creation forms (**newTag**), exception introduction (**injx**) and elimination (**casex**) forms, and exception raising (**raise**) and handling (**handle**) forms. Terms are identified modulo reordering of fields in records, bindings in **letrec** forms, and clauses in $\mathbf{case}^{+}$ and **casex** forms. Records must not have two fields with the same name and **letrec** forms must not have two bindings with the same name. The **newTag** form handles SML exception generativity by returning a new exception tag each time it is evaluated; these tags are used by the exception introduction and elimination forms. The set **Primop** of primitive operators includes standard operators on base types as well as operators for manipulating reference cells (**mkRef**, **getRef**, **setRef**) and arrays (not shown).

The set **UntValContext** is a subset of **UntContext** designated as *syntactic values*. These are a conservative approximation of the *semantic values*, those terms whose evaluation yields a value (and thus does not go wrong or raise an exception) without allocating/inspecting/setting a reference cell for any possible evaluation environment. The **letrec** form must bind variables only to syntactic values. This restriction is sufficient to allow defining a semantics for **letrec** that is consistent with call-by-value evaluation. The restriction is more lenient than in SML, where **let val rec** only allows manifest abstractions. This lenience

Untyped Syntax

$$x, y, z \in \ \mathbf{Variable} \qquad c \in \mathbf{Constant} \qquad f \in \mathbf{Field} \qquad n \in \mathbf{Nat} = \{0, 1, 2, \ldots\} \qquad p \in \mathbf{Pos} = \mathbf{Nat} - \{0\}$$

$$\begin{array}{lll}
pr \in \mathbf{Primop} & ::= & + \mid < \mid \mathbf{mkRef} \mid \mathbf{getRef} \mid \mathbf{setRef} \mid \ldots \\
\hat{C} \in \mathbf{UntContext} & ::= & \square \mid c \mid x \mid pr(\hat{C}_i)_{i=1}^n \mid \lambda x.\hat{C} \mid \hat{C}_1 \mathbin{@} \hat{C}_2 \mid \mathbf{let}\ x = \hat{C}_1\ \mathbf{in}\ \hat{C}_2 \mid \mathbf{letrec}\ \{x_i = \hat{D}_i\}_{i=1}^n\ \mathbf{in}\ \hat{C} \\
& & \mid \times(f_i = \hat{C}_i)_{i=1}^n \mid \pi_f^\times\ \hat{C} \mid \iota_f^+\ \hat{C} \mid \mathbf{case}^+\ \hat{C}_0\ \mathbf{bind}\ x\ \mathbf{in}\ \{f_i \Rightarrow C_i\}_{i=1}^p \\
& & \mid \mathbf{newTag} \mid \mathbf{injx}(\hat{C}_1, \hat{C}_2) \mid \mathbf{casex}\ \hat{C}_0\ \mathbf{bind}\ x\ \mathbf{in}\ \{\hat{C}_i \Rightarrow \hat{C}_i'\}_{i=1}^n\ \mathbf{else}\ \hat{C}_0' \\
& & \mid \mathbf{raise}\ \hat{C} \mid \hat{C}_1\ \mathbf{handle}\ x\ \mathbf{in}\ \hat{C}_2 \\
\hat{D} \in \mathbf{UntValContext} & ::= & \square \mid c \mid \lambda x.\hat{C} \mid \times(f_i = \hat{D}_i)_{i=1}^n \mid \iota_f^+\ \hat{D} \mid \mathbf{let}\ x = \hat{D}_1\ \mathbf{in}\ \hat{D}_2 \mid \mathbf{let}\ x = \hat{D}\ \mathbf{in}\ x \\
& & \mid \mathbf{letrec}\ \{x_i = \hat{D}_i\}_{i=1}^n\ \mathbf{in}\ \hat{D} \mid \mathbf{letrec}\ \{x_i = \hat{D}_i\}_{i=1}^n\ \mathbf{in}\ x_j \quad \text{where } 1 \le j \le n \\
\hat{M}, \hat{N} \in \mathbf{UntTerm} & = & \{\, \hat{C} \mid \square \text{ does not occur in } \hat{C} \,\} \\
\hat{V} \in \mathbf{UntValue} & = & \{\, \hat{D} \mid \square \text{ does not occur in } \hat{D} \,\}
\end{array}$$

Syntax Shared between Types and Terms

$$l, k \in \mathbf{Label} = \mathbf{Nat} \qquad \varnothing \neq \phi, \psi \subset \mathbf{Label}$$

Types

$$\begin{array}{lll}
o \in \mathbf{BaseType} & ::= & \mathbf{unit} \mid \mathbf{int} \mid \mathbf{char} \mid \mathbf{real} \mid \ldots \\
\alpha \in \mathbf{TypeVariable} & & \\
\xi \in \mathbf{GuardedType} & ::= & o_\psi^\phi \mid \upsilon_1 \xrightarrow[\psi]{\phi} \upsilon_2 \mid \times_\psi^\phi\{f_i : \upsilon_i\}_{i=1}^n \mid \wedge_\psi^\phi\{f_i : \upsilon_i\}_{i=1}^p \mid +_\psi^\phi\{f_i : \upsilon_i\}_{i=1}^p \mid \vee_\psi^\phi\{f_i : \upsilon_i\}_{i=1}^p \\
& & \mid \mathbf{ref}_\psi^\phi[\upsilon] \mid \mathbf{exn} \mid \mathbf{xtag}_\psi^\phi[\upsilon] \mid \ldots \\
\upsilon \in \mathbf{UnguardedType} & ::= & \alpha \mid \mathbf{tletrec}\ \{\alpha_i = \xi_i\}_{i=1}^n\ \mathbf{in}\ \upsilon \mid \xi \\
\sigma, \tau \in \mathbf{Type} & = & \{\xi \mid \mathrm{FV}(\xi) = \varnothing\}
\end{array}$$

Type-Annotated Contexts

$$\begin{array}{lll}
C \in \mathbf{Context} & ::= & \square^\tau \mid c_\psi^l \mid x^\tau \mid pr\,(R)\,((K_i)\,C_i)_{i=1}^n \mid \lambda_\psi^l x^\tau.C \mid C_1 \mathbin{@_k^\phi} C_2 \\
& & \mid \mathbf{let}\ x^\tau = C_1\ \mathbf{in}\ C_2 \mid \mathbf{letrec}\ \{x_i^{\tau_i} = C_i\}_{i=1}^n\ \mathbf{in}\ C \mid \mathbf{coerce}\,(\sigma, \tau)\,C \\
& & \mid \times_\psi^l(f_i = C_i)_{i=1}^n \mid \pi_f^\times \binom{\phi}{k} C \mid (\iota_f^+ \binom{l}{\psi} C)^\tau \mid \mathbf{case}^+ \binom{\phi}{k} C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (f_i \Rightarrow^{\tau_i} C_i)_{i=1}^p \\
& & \mid \wedge_\psi^l(f_i = C_i)_{i=1}^p \mid \pi_f^\wedge \binom{\phi}{k} C \mid (\iota_f^\vee \binom{l}{\psi} C)^\tau \mid \mathbf{case}^\vee \binom{\phi}{k} C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (f_i \Rightarrow^{\tau_i} C_i)_{i=1}^p \\
& & \mid \mathbf{newTag}^\tau\binom{l}{\psi} \mid \mathbf{injx}_k^\phi(C_1, C_2) \mid \mathbf{casex}\ C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (C_i \binom{\phi_i}{k_i} \Rightarrow^{\tau_i} C_i')_{i=1}^n\ \mathbf{else}\ C_0' \\
& & \mid \mathbf{raise}\ C \mid C_1\ \mathbf{handle}\ x\ \mathbf{in}\ C_2
\end{array}$$

$$R \in \mathbf{SourceBundle} \ ::= \ \bullet \mid {}^{l}_{\psi} \qquad\qquad K \in \mathbf{SinkBundle} \ ::= \ \circ \mid {}^{\phi}_{k}$$

Type Erasure (a partial function from **Context** to **UntContext**)

$$|\square^\tau| \equiv \square \qquad \left|c_\psi^l\right| \equiv c \qquad |x^\tau| \equiv x \qquad |\mathbf{coerce}\,(\sigma, \tau)\,C| \equiv |C|$$

$$\left|pr\,(R)\,((K_i)\,C_i)_{i=1}^n\right| \equiv pr(|C_i|)_{i=1}^n \qquad |\mathbf{let}\ x^\tau = C_1\ \mathbf{in}\ C_2| \equiv \mathbf{let}\ x = |C_1|\ \mathbf{in}\ |C_2|$$

$$\left|\lambda_\psi^l x^\tau.C\right| \equiv \lambda x.|C| \qquad \left|C_1 \mathbin{@_k^\phi} C_2\right| \equiv |C_1| \mathbin{@} |C_2|$$

$$\left|\mathbf{letrec}\ \{x_i^{\tau_i} = C_i\}_{i=1}^n\ \mathbf{in}\ C\right| \equiv \mathbf{letrec}\ \{x_i = |C_i|\}_{i=1}^n\ \mathbf{in}\ |C|, \quad \text{if } \forall_{i=1}^n.\ |C_i| \in \mathbf{UntValContext}$$

$$\left|\times_\psi^l(f_i = C_i)_{i=1}^n\right| \equiv \times(f_i = |C_i|)_{i=1}^n \qquad \left|\wedge_\psi^l(f_i = C_i)_{i=1}^p\right| \equiv \begin{cases} \hat{D} & \text{if } \hat{D} \equiv |C_1| \equiv \cdots \equiv |C_p|\,, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\left|\pi_f^\times \binom{\phi}{k} C\right| \equiv \pi_f^\times\ |C| \qquad \left|\pi_f^\wedge \binom{\phi}{k} C\right| \equiv |C| \qquad \left|(\iota_f^+ \binom{l}{\psi} C)^\tau\right| \equiv \iota_f^+\ |C| \qquad \left|(\iota_f^\vee \binom{l}{\psi} C)^\tau\right| \equiv |C|$$

$$\left|\mathbf{case}^+ \binom{\phi}{k} C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (f_i \Rightarrow^{\tau_i} C_i)_{i=1}^p\right| \equiv \mathbf{case}^+\ |C_0|\ \mathbf{bind}\ x\ \mathbf{in}\ \{f_i \Rightarrow |C_i|\}_{i=1}^p$$

$$\left|\mathbf{case}^\vee \binom{\phi}{k} C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (f_i \Rightarrow^{\tau_i} C_i)_{i=1}^p\right| \equiv \begin{cases} \mathbf{let}\ x = |C_0|\ \mathbf{in}\ |C_1| & \text{if } |C_1| \equiv \cdots \equiv |C_p|\,, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\left|\mathbf{casex}\ C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (C_i \binom{\phi_i}{k_i} \Rightarrow^{\tau_i} C_i')_{i=1}^n\ \mathbf{else}\ C_0'\right| \equiv \mathbf{casex}\ |C_0|\ \mathbf{bind}\ x\ \mathbf{in}\ \{|C_i| \Rightarrow |C_i|'\}_{i=1}^n\ \mathbf{else}\ |C_0|'$$

$$\left|\mathbf{newTag}^\tau\binom{l}{\psi}\right| \equiv \mathbf{newTag} \qquad \left|\mathbf{injx}_k^\phi(C_1, C_2)\right| \equiv \mathbf{injx}(|C_1|\,, |C_2|)$$

$$|\mathbf{raise}\ C| \equiv \mathbf{raise}\ |C| \qquad \left|\hat{C}_1\ \mathbf{handle}\ x\ \mathbf{in}\ \hat{C}_2\right| \equiv |C_1|\ \mathbf{handle}\ x\ \mathbf{in}\ |C_2|$$

Type-Annotated Terms and Values

$$M, N \in \mathbf{Term} = \{C \mid \text{the type erasure } |C| \in \mathbf{UntTerm}\}$$

$$V \in \mathbf{Value} = \{C \mid \text{the type erasure } |C| \in \mathbf{UntValue}\}$$

Figure 6: Syntax of CIL.

Typing Rules

$$(\text{hole})\ \frac{}{\Gamma \vdash \square^{\tau} : \tau} \qquad (\text{const})\ \frac{\mathrm{ConstType}(c) = o}{\Gamma \vdash c^{l}_{\psi} : o^{\{l\}}_{\psi}} \qquad (\text{var})\ \frac{}{\Gamma + \{x{:}\tau\} \vdash x^{\tau} : \tau}$$

$$(\text{primapp})\ \frac{\forall^{n}_{i=0}.\ \Gamma \vdash C_i : \tau_i;\ \mathrm{PrimType}_n(pr, K_1, \tau_1, \cdots, K_n, \tau_n, R, \tau)}{\Gamma \vdash pr\,(R)\,((K_i)\,C_i)^{n}_{i=1} : \tau}$$

$$(\text{letrec})\ \frac{\forall^{n}_{i=0}.\ (\Gamma + \{x_j{:}\tau_j\}^{n}_{j=1} \vdash C_i : \tau_i);\ \forall^{n}_{i=1}.\ (|C_i| \equiv \hat{D}_i)}{\Gamma \vdash \mathbf{letrec}\ \{x_i^{\tau_i} = C_i\}^{n}_{i=1}\ \mathbf{in}\ C_0 : \tau_0}$$

$$(\text{coerce})\ \frac{\Gamma \vdash C : \sigma;\ \sigma \leq \tau}{\Gamma \vdash \mathbf{coerce}\,(\sigma, \tau)\,C : \tau} \qquad (\text{let})\ \frac{\Gamma + \{x{:}\sigma\} \vdash C_2 : \tau;\ \Gamma \vdash C_1 : \sigma}{\Gamma \vdash \mathbf{let}\ x^{\sigma} = C_1\ \mathbf{in}\ C_2 : \tau}$$

$$(\to \text{intro})\ \frac{\Gamma + \{x{:}\sigma\} \vdash C : \tau}{\Gamma \vdash \lambda^{l}_{\psi} x^{\sigma}.C : \sigma \xrightarrow[\psi]{\{l\}} \tau} \qquad (\to \text{elim})\ \frac{\Gamma \vdash C_1 : \sigma \xrightarrow[\{k\}]{\phi} \tau;\ \Gamma \vdash C_2 : \sigma}{\Gamma \vdash C_1\ @^{\phi}_{k}\ C_2 : \tau}$$

$$(\times \text{intro})\ \frac{\forall^{n}_{i=1}.\ \Gamma \vdash C_i : \tau_i}{\Gamma \vdash \times^{l}_{\psi}(f_i = C_i)^{n}_{i=1} : \times^{\{l\}}_{\psi}\{f_i : \tau_i\}^{n}_{i=1}} \qquad (\wedge \text{intro})\ \frac{|C_1| \equiv \cdots \equiv |C_n| \equiv \hat{D};\ \forall^{p}_{i=1}.\ \Gamma \vdash C_i : \tau_i}{\Gamma \vdash \wedge^{l}_{\psi}(f_i = C_i)^{p}_{i=1} : \wedge^{\{l\}}_{\psi}\{f_i : \tau_i\}^{p}_{i=1}}$$

$$(\times \text{elim})\ \frac{\Gamma \vdash C : \times^{\phi}_{\{k\}}\{f_j : \tau_j\}^{n}_{j=1};\ 1 \leq i \leq n}{\Gamma \vdash \pi^{\times}_{f_i}\ \binom{\phi}{k}\ C : \tau_i} \qquad (\wedge \text{elim})\ \frac{\Gamma \vdash C : \wedge^{\phi}_{\{k\}}\{f_j : \tau_j\}^{p}_{j=1};\ 1 \leq i \leq p}{\Gamma \vdash \pi^{\wedge}_{f_i}\ \binom{\phi}{k}\ C : \tau_i}$$

$$(+ \text{intro})\ \frac{\Gamma \vdash C : \tau_i;\ 1 \leq i \leq p;\ \sigma \equiv +^{\{l\}}_{\psi}\{f_j : \tau_j\}^{p}_{j=1}}{\Gamma \vdash (\iota^{+}_{f_i}\ \binom{l}{\psi}\ C)^{\sigma} : \sigma} \qquad (\vee \text{intro})\ \frac{\Gamma \vdash C : \tau_i;\ 1 \leq i \leq p;\ \sigma \equiv \vee^{\{l\}}_{\psi}\{f_j : \tau_j\}^{p}_{j=1}}{\Gamma \vdash (\iota^{\vee}_{f_i}\ \binom{l}{\psi}\ C)^{\sigma} : \sigma}$$

$$(+ \text{elim})\ \frac{\Gamma \vdash C_0 : +^{\phi}_{\{k\}}\{f_i : \tau_i\}^{p}_{i=1};\ \forall^{p}_{i=1}.\ \Gamma + \{x{:}\tau_i\} \vdash C_i : \tau}{\Gamma \vdash \mathbf{case}^{+}\ \binom{\phi}{k}\ C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (f_i \Rightarrow^{\tau_i} C_i)^{p}_{i=1} : \tau}$$

$$(\vee \text{elim})\ \frac{\Gamma \vdash C_0 : \vee^{\phi}_{\{k\}}\{f_i : \tau_i\}^{p}_{i=1};\ \forall^{p}_{i=1}.\ \Gamma + \{x{:}\tau_i\} \vdash C_i : \tau;\ \ |C_1| \equiv \cdots \equiv |C_n|}{\Gamma \vdash \mathbf{case}^{\vee}\ \binom{\phi}{k}\ C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (f_i \Rightarrow^{\tau_i} C_i)^{p}_{i=1} : \tau}$$

$$(\text{xtag})\ \frac{}{\Gamma \vdash \mathbf{newTag}^{\tau}\binom{l}{\psi} : \mathbf{xtag}^{\{l\}}_{\psi}[\tau]} \qquad (\mathbf{exn}\ \text{intro})\ \frac{\Gamma \vdash C_1 : \mathbf{xtag}^{\phi}_{\{k\}}[\tau];\ \Gamma \vdash C_2 : \tau}{\Gamma \vdash \mathbf{injx}^{\phi}_{k}(C_1, C_2) : \mathbf{exn}}$$

$$(\mathbf{exn}\ \text{elim})\ \frac{\Gamma \vdash C_0 : \mathbf{exn};\ \Gamma \vdash C'_0 : \tau;\ \forall^{n}_{i=1}.\ (\Gamma \vdash C_i : \mathbf{xtag}^{\phi_i}_{\{k_i\}}[\tau_i];\ \Gamma + \{x{:}\tau_i\} \vdash C'_i : \tau)}{\Gamma \vdash \mathbf{casex}\ C_0\ \mathbf{bind}\ x\ \mathbf{in}\ (C_i\ \binom{\phi_i}{k_i} \Rightarrow^{\tau_i} C'_i)^{n}_{i=1}\ \mathbf{else}\ C'_0 : \tau}$$

$$(\text{raise})\ \frac{\Gamma \vdash C : \mathbf{exn}}{\Gamma \vdash \mathbf{raise}\ C : \tau} \qquad (\text{handle})\ \frac{\Gamma \vdash C_1 : \tau;\ \Gamma + \{x{:}\mathbf{exn}\} \vdash C_2 : \tau}{\Gamma \vdash C_1\ \mathbf{handle}\ x\ \mathbf{in}\ C_2 : \tau}$$

Primop Type Relations (a few example relation members)

$$\mathrm{PrimType}_2(+, {}^{\phi_1}_{k_1}, \mathbf{int}^{\phi_1}_{\{k_1\}}, {}^{\phi_2}_{k_2}, \mathbf{int}^{\phi_2}_{\{k_2\}}, {}^{l}_{\psi}, \mathbf{int}^{\{l\}}_{\psi}) \qquad \mathrm{PrimType}_1(\mathbf{mkRef}, \circ, \tau, {}^{l}_{\psi}, \mathbf{ref}^{\{l\}}_{\psi}[\tau])$$

$$\mathrm{PrimType}_1(\mathbf{getRef}, {}^{\phi}_{k}, \mathbf{ref}^{\phi}_{\{k\}}[\tau], \bullet, \tau) \qquad \mathrm{PrimType}_2(\mathbf{setRef}, {}^{\phi}_{k}, \mathbf{ref}^{\phi}_{\{k\}}[\tau], \circ, \tau, {}^{l}_{\psi}, \mathbf{unit}^{\{l\}}_{\psi})$$

$$\mathrm{PrimType}_2(<, {}^{\phi_1}_{k_1}, \mathbf{int}^{\phi_1}_{\{k_1\}}, {}^{\phi_2}_{k_2}, \mathbf{int}^{\phi_1}_{\{k_1\}}, {}^{l}_{\psi}, \times^{\{l\}}_{\psi}\{\mathrm{true} : \mathbf{unit}^{\phi'}_{\psi'}, \mathrm{false} : \mathbf{unit}^{\phi''}_{\psi''}\})$$

Subtyping Rules (a few example rules)

$$(\to\leq)\ \frac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\sigma \xrightarrow[\psi]{\phi} \tau \leq \sigma \xrightarrow[\psi']{\phi'} \tau} \qquad (\times\leq)\ \frac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\times^{\phi}_{\psi}\{f_i : \tau_i\}^{n}_{i=1} \leq \times^{\phi'}_{\psi'}\{f_i : \tau_i\}^{n}_{i=1}} \qquad (\mathbf{ref}\leq)\ \frac{\phi \subseteq \phi';\ \psi' \subseteq \psi}{\mathbf{ref}^{\phi}_{\psi}[\tau] \leq \mathbf{ref}^{\phi'}_{\psi'}[\tau]}$$

Figure 7: Typing rules of CIL.

provides more flexibility in the CIL compiler, e.g., allowing the closures for recursively defined functions to be recursive records.

The type syntax includes types for primitive data (o), functions ($\rightarrow$), real records ($\times$), virtual records ($\wedge$), real variants ($+$), virtual variants ($\vee$), reference cells (**ref**), exceptions (**exn**), and exception tags (**xtag**). All types except **exn** are annotated with a *flow bundle* $\substack{\phi \\ \psi}$, where ϕ is a set of *source labels* approximating the definition sites for values having the type, and ψ is a set of *sink labels* approximating the use sites for values having the type. We treat **tletrec** forms as equivalent to their infinite unwindings. For this to be sensible, the bindings in a **tletrec** form are required to be elements of **GuardedType**, which guarantees that all bound type variables appear underneath a type constructor other than **tletrec**.

Type-annotated contexts and terms are decorated with type and flow information. In addition to type- and flow-annotated versions of the untyped terms, there are some additional constructs at the type-annotated level: introduction ($\wedge$) and elimination ($\pi^{\wedge}$) forms for virtual records, introduction ($\iota^{\vee}$) and elimination ($\mathbf{case}^{\vee}$) forms for virtual variants, and coercion forms (**coerce**) that perform explicit subtyping. The components of a virtual record and the clauses of a virtual **case** expression are required to be the same modulo type and flow annotations. This requirement is formalized by the definition of the type erasure function, $|\,.\,|$, which maps type-annotated contexts to untyped contexts. Note that the bindings of a type-annotated **letrec** term and the components of a virtual record introduction term must type erase to syntactic values. The latter restriction is similar to the value restriction for polymorphism in SML.

Type-annotated primitive applications are decorated with an optional sink bundle for each operand position and an optional source bundle. This allows distinguishing operand positions that act as sinks (i.e., use the argument value) from those that do not, and distinguishing primitives that act as sources (i.e., generate new values) from those that return existing values. For example: applications of arithmetic operators like $+$ are sinks for all operands and a source for the resulting value; **mkRef** is a source of a reference cell but not a sink for its argument; **getRef** is a sink for the reference cell, but is not a source of the return value; and **setRef** is a sink for the reference cell operand, but is not a sink for the value that is the new cell contents.

The type-annotated syntax is designed so that well typed terms are isomorphic to typing derivation trees generated by the typing rules in figure 7. The (primapp) rule uses the PrimType relation, which for each primitive operator encodes knowledge of the operand and result types as well as which operand positions are sinks and whether the operator acts as a source. Representative clauses of the definition of PrimType relation are given in figure 7. The (coerce) rule uses a "shallow" subtyping relation $\leq$ that allows adding source labels and removing sink labels but requires any component types to be invariant. The shallow subtyping restriction facilitates type-based program transformation in our framework; see [WDMT0X] for a discussion.