Simple Usage Polymorphism

Keith Wansbrough*
Computer Laboratory
University of Cambridge
Cambridge CB2 3QG, England
kw217@cl.cam.ac.uk
www.cl.cam.ac.uk/users/kw217/

Simon Peyton Jones
Microsoft Research Ltd
St George House, 1 Guildhall St
Cambridge CB2 3NH, England
simonpj@microsoft.com
research.microsoft.com/Users/simonpj/

Abstract

We present a novel inference algorithm for a type system featuring subtyping and usage (annotation) polymorphism. This algorithm infers *simply-polymorphic* types rather than the constrained-polymorphic types usual in such a setting; it achieves this by means of *constraint approximation.* The algorithm is motivated by practical considerations and experience of a previous system, and has been implemented in a production compiler with positive results. We believe the algorithm may well have applications in settings other than usage-type inference.

1 Introduction

Consider the following function call, written in the lazy functional language Haskell:

$$f\ (g\ x)$$

Most Haskell implementations will construct a self-updating *thunk*, or suspension, for f's argument, $(g\ x)$, and pass that thunk to f. If it can be demonstrated that f uses its argument at most once, then this thunk need not be overwritten after the evaluation of $(g\ x)$. The thunk is still constructed, but it is less expensive than before. It turns out that this is but one of a range of motivations for analysing functional programs (strict ones as well as lazy ones) to figure out when a sub-expression will only be "used once".

Thus motivated, several researchers including ourselves have sought a type system that can statically determine some notion of used-once-ness. In an earlier paper we described just such a system [WPJ99], *UsageSP*, focusing especially on two points: we dealt with a realistic language, including polymorphism and recursive data types; and we used subtyping to make *UsageSP* more expressive than an earlier, monomorphic proposal [TWM95].

At that time we had only a prototype implementation. To our dismay, when we scaled it up to a full scale system, we discovered that *UsageSP* was almost useless in practice! It turned out that curried functions received rather poor types; since currying is ubiquitous in Haskell, this meant that the system missed almost all used-once expressions, including ones that are blatantly obvious to the programmer. We explain the problem in detail in Section 3.2.

This paper presents our solution to the problem of currying. Specifically, our new contributions are these:

- We present the results of comprehensive measurements of the potential gains to be had from a system like *UsageSP*. These results confirm those of an earlier study [Mar93], and of folklore: in lazy languages, a very high proportion of thunks are only demanded once (Section 2).
- Observing (along with Gustavsson [Gus98] and contrary to our previous belief) that subtyping is not sufficient to give satisfactory types, we introduce usage polymorphism and characterise precisely when it is necessary; namely, when a usage variable appears in both a covariant and a contravariant position in a function's type (Section 3.3). This form of polymorphism differs from the *polyvariance* discussed as future work in [TWM95] and [Gus98], in that our system provides a usage-polymorphic type for a *single copy* of a function, without specialisation or cloning.
- We argue that the full generality of constrained polymorphism, while technically straightforward, has a very poor power-to-weight ratio (Section 3.4). Our main conceptual contribution is to identify a "sweet spot" in the design space: the types it gives are much smaller, albeit not quite as refined, as those of a constrained polymorphic system (Section 3.5).
- Our principal technical contribution is a new type inference algorithm that infers types that are just polymorphic enough to express these co-/contra- dependencies (Section 6). So far as we know, this algorithm is novel, and it may well have applications in other settings, such as binding-time analysis [DHM95].
- We sketch our implementation, which is part of the Glasgow Haskell Compiler (GHC), a state of the art compiler for Haskell (Section 7). GHC uses a Girard–Reynolds-style typed intermediate language. We extend this language with usage abstractions and applications, so that the result of usage-type inference is expressed in the intermediate language itself (Section 5). As a result, subsequent program transformations can preserve accurate usage types.

*Research supported in part by the Commonwealth Scholarship Commission under the Commonwealth Scholarship and Fellowship Plan, and by UK EPSRC grant GR/M04716.

Appears in informal proceedings of *Third International Workshop on Types in Compilation (TIC 2000),* 21 September 2000, Montreal, Canada.

- We give preliminary measurements of the effectiveness of the new algorithm (Section 7.1). It successfully deals with the currying problem, although the bottom-line results are still a bit disappointing. However, we have evidence that they can be improved substantially without changing the type system.

Our type system has just two annotations: "used once" (*i.e.*, at most once) and "used many times" (*i.e.*, don't know). We are confident that we can smoothly extend this little two-point space into a seven-point lattice (see Section 9) that will combine usage analysis, strictness analysis and absence analysis, thereby combining three separate analyses in one uniform framework. This paper represents solid progress towards that alluring goal.

2 The opportunity

In our earlier paper we described in some detail the benefits that accrue from static knowledge that a value is used at most once [WPJ99]. Avoiding updates for thunks is an obvious benefit, but others include better inlining and let-floating. We will not repeat this discussion here.

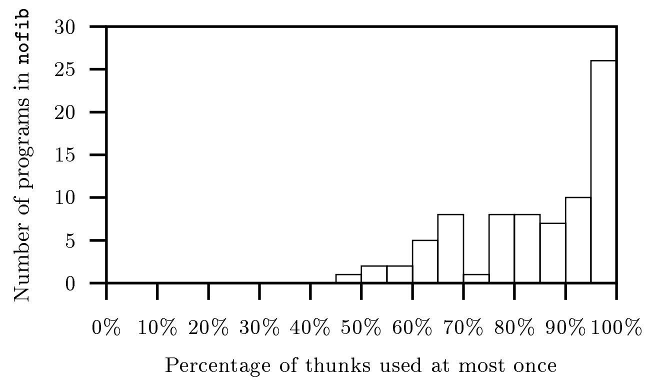
So knowledge of used-once-ness is certainly beneficial – but *how* beneficial? If the maximum benefit is tiny, there is little point in working on the analysis. It is hard to quantify the potential benefits of making other transformations (inlining, let-floating) more widely applicable, but we can directly quantify the maximum number of thunks whose update could be avoided.

We modified our Haskell compiler, GHC, so that it generated code to record the proportion of all allocated thunks that are demanded at most once. The necessary modifications are rather simple: we only need to track the total number of thunks allocated and, for each thunk, whether it is entered never, once, or more than once. We compiled and ran the entire `nofib` suite, a large suite of around 50 benchmark programs ranging from tiny ones up to 10,000 line applications [Par93].

Figure 1 gives the results, which amply support the folklore and are consistent with the data of [Mar93]. In *every program but one,* the majority of thunks are demanded at most once, and hence do not need to be updated. In more than a third of the programs, over 95% of thunks are never entered more than once! There is clearly huge scope for optimisation here.

Of course, no static analysis will find *all* thunks that are demanded only once, but at least we now know for sure that our analysis is digging in rich soil.

Figure 1 The opportunity.



3 The problem of currying

In this section we will identify the problem we propose to attack, beginning with a quick review of the context.

3.1 Usage types

Our compiler takes an ordinary Haskell program and begins by performing type inference on it, translating it into an explicitly-typed intermediate language, the *GHC Core language.* This Core language is a Girard–Reynolds-style lambda calculus, augmented with let, letrec, constructors, case expressions, and primitive operations. We then perform usage-type inference on this program, yielding a new Core program whose types embody usage information. This information is used to guide the compiler's transformations. At any time, the usage types can be refreshed by re-running the inference algorithm. The programmer never sees the usage types.

For example, consider the Core definition:

$$f = \lambda x \,.\, x + 1$$

Initially f has the type $\mathsf{Int} \to \mathsf{Int}$. After usage inference it gets the more informative type $(\mathsf{Int}^1 \to \mathsf{Int}^\omega)^\omega$. This type says that (a) f demands its argument at most once (Int^1), (b) that its result may be used arbitrarily often (Int^ω) and, (c) that the function itself may be used arbitrarily often (the outer ω).

f's type says that it *may* be applied to an argument of type Int^1, but we employ subtyping to permit f to be applied to an argument of type Int^ω. This subtyping directly matches our operational intuitions: a function can accept any argument which can be used more often than the function requires. The use of subtyping in this way was a major feature of our earlier paper,[1] and we do not stress it further here [WPJ99].

So much for the argument type of f. What about (c), the outermost "ω" in f's type? This part of f's type is affected by the way in which f is used, not by f's definition. For example, suppose that f's definition had appeared in this context:

$$\begin{array}{l} \mathsf{let}\ f = \lambda x \,.\, x + 1 \\ \mathsf{in}\ \ \mathsf{if}\, b \\ \quad\quad \mathsf{then}\ f\ 1 \\ \quad\quad \mathsf{else}\ \ f\ 2 \end{array}$$

Now, it is clear that f is called only once, so f could have been given the type $(\mathsf{Int}^1 \to \mathsf{Int}^\omega)^1$. (Actually, such an improvement has no operational benefit in this case, but in principle it might.) However, if f is a top-level (exported) definition in a module, called from arbitrary places

[1] This use of subtyping turns out to have been proposed independently by Gustavsson [Gus98] and by Faxén [Fax95]; in the context of binding-time analysis it was already well-known [Mos93].

elsewhere, we pessimistically assume that it is called many times, and force its outermost annotation to ω.

3.2 Currying

Now consider the following innocuous definition:

$$g = \lambda x \,.\, \lambda y \,.\, x + y - 1$$

It would seem as if g demands its arguments only once, so we would expect a type like

$$g :: (\mathsf{Int}^1 \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^\omega)^\omega$$

The trouble is that this type is simply not correct. Suppose g was used like this:

$$\begin{array}{ll} \mathsf{let} & h = g\ \mathit{wurble} \\ \mathsf{in} & h\,3 + h\,4 \end{array}$$

How many times will the thunk *wurble* be demanded? Twice, once for each call of h! So g's type must be wrong. Going back to the definition of g, we see that x is used inside the λy-abstraction, so x is used at least as much as the λy-abstraction. There are two correct types we can assign to g (we subscript the g so that we can name these different typings):

$$\begin{array}{l} g_2 :: (\mathsf{Int}^\omega \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^\omega)^\omega \\ g_3 :: (\mathsf{Int}^1 \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^1)^\omega \end{array}$$

In the first, g_2's first argument is given the (bad) annotation ω, but in exchange the function returned by g *wurble* has a ω-type. In the second, g_3's first argument has the (good) annotation 1, but the price is that the partial application of g cannot be shared.

These constraints are perfectly reasonable: if g *wurble* is called many times, then *wurble* will be demanded many times. What is *not* reasonable is that the analysis of [WPJ99] is forced to make the choice once and for all, at g's definition site. If g is *ever* partially applied, then *all* uses of g will have an ω annotation on their first argument. The frequency of partial applications in Haskell code makes this a very poor strategy.

3.3 The solution: Simple usage polymorphism

Now that the problem is clear, the solution is completely obvious. We want usage polymorphism, thus:

$$g_4 :: (\forall u \,.\, \mathsf{Int}^u \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^u)^\omega$$

Now the two valid types are instances of this polymorphic type, and each call site can instantiate g_4's type appropriately. (The reader may be surprised by the location of the $\forall u$ *inside* the outermost usage annotation; see Section 5.1 for a discussion of this point.)

That solves the problem nicely. The remaining challenge is how to come up with this type for g. After all, here is another possible type for g:

$$g_5 :: (\forall u, v, x \,.\, \mathsf{Int}^u \to (\mathsf{Int}^v \to \mathsf{Int}^x)^u)^\omega$$

Here, we have replaced the ω and 1 annotations with usage variables, and then quantified over them. This type is actually sound in the type system we define in Section 5, but it is unnecessarily complicated, because *nothing is gained by the extra polymorphism.* For example, the usage "v" on the second argument of g_5 does not give any extra information to the caller ("1" was as informative as possible), nor does it make g_5 any more applicable. Similarly, the usage "x" instead of "ω" carries no benefit. This is a key point of our earlier paper [WPJ99, §6.3].

We can characterise the situation quite precisely, as follows:

- A usage variable in a covariant (positive) position – for example, x in the type of g_5 – may as well be turned into the constant "ω".
- A usage variable in a contravariant (negative) position – for example, v in the type of g_5 – may as well be turned into the constant "1".
- Only usage variables that appear both covariantly and contravariantly in the function's type must be universally quantified.

This argument depends on the use of subtyping. One might argue that it would be easier to get rid of subtyping and use polymorphism instead. But in our explicitly-typed intermediate language, at every call site of a polymorphic function, the function is applied to all the type and usage arguments necessary to instantiate all its universally quantified variables. So the more variables we quantify over, the larger our intermediate programs will become. Further, there are two distinct properties to be expressed. Our system therefore uses *subtyping* to express the fact that a function may accept a range of argument types, and *polymorphism* to express interdependencies between the usage annotations within a single type.

Simple usage polymorphism, therefore, describes a type system that supports subtyping and polymorphism, but does not permit subtyping constraints on quantified variables.

It should be noted that while simple usage polymorphism is certainly better than the monomorphic system of [WPJ99], it doesn't solve all our problems. Binary functions like g now get optimal types, but ternary functions like *plus3* (Figure 2) end up unifying the usages of their first two arguments, potentially forcing thunks in fact used at most once to be annotated ω. Ultimately, polyvariance may be the best solution, and we discuss this further in Section 7.2.

3.4 Constrained polymorphism

Have we now gone far enough? From a technical standpoint, a more natural choice might seem to be *constrained polymorphism* [Cur90, AW93].[2] A yet more general type for g is this:

$$g_6 :: (\forall u, v : u \leq v \,.\, \mathsf{Int}^u \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^v)^\omega$$

This type makes explicit that the usage of g's first argument and its partial application don't need to be the same; all that is needed is that the usage of the argument is at least that of the partial application. (The constraint may look back to front; we will explain the polarity of our subtyping relationship in Section 5.1.)

[2] An intermediate possibility between simple and constrained polymorphism, considered no further here, might be *bounded* polymorphism, as introduced by Cardelli [CW85, CMMS94].

Table 1 Usage type system design space.

	Monomorphic	Simple polymorphism	Constrained polymorphism
No subtyping	[TWM95]	(2)	(3)
Fun arg subtyping	[WPJ99]$^{(4)}$	This paper	[GS00]$^{(6)}$
	(cheap enough)		(expressive enough)

Constrained polymorphism has been moderately well studied [TS96, OSW98, PS96, JP99], and it is certainly more expressive than simple polymorphism (the simplest type on which they differ is that of *plus3*; see Section 6.3). However, the practical costs are heavy. Types become (much) larger, because there are typically many more quantified variables, together with many constraints connecting them [Pot98, AWP97]. Furthermore, GHC's entire internal language would have to be augmented with type constraints, no simple matter.

3.5 Summary

Table 1 summarises our conclusions so far. The two axes of the table correspond to two different design choices. Systems in the top row lack subtyping, while those in the bottom row allow subtyping at function applications. Systems in the first column are monomorphic in usage annotations (polymorphism in the underlying language is an orthogonal issue, as we showed in [WPJ99]). Systems in the middle column offer simple usage polymorphism, while those in the third column employ constrained polymorphism. (3), for example, is suggested in [TWM95, §4.5]. [Gus98] and [Fax95] could also be placed in (4), and [BS96] (for uniqueness, rather than usage) and perhaps [Fax97] belong in (6).

Our central hypothesis is this: The analyses in the first column, and (2) in the second, are not expressive enough; the third column is unreasonably expensive; thus the combination of simple usage polymorphism and subtyping offers a fine compromise between expressiveness and cost. It seems clear that the former are certainly insufficiently expressive; as to the latter the vast body of work in the area of constrained polymorphism already cited confirms its high cost, although this may well be justified by the benefits (*q.v.* [GS00]). The present paper proposes a novel way of trading off these costs against the benefits.

Notice that though the *type system* is simpler without constrained polymorphism, the *inference algorithm* may be more complex. Any inference algorithm is going to work by gathering and then resolving constraints. With constrained polymorphism, we can quantify over these constraints; with simple polymorphism we must instead approximate. Specifically, where we have a constraint $u \leq v$ we simply identify (unify) u and v (compare the types for g in Sections 3.3 and 3.4). That's the idea anyway; the actual algorithm is quite subtle, as we discuss in Section 6.2, and constitutes the technical heart of our paper.

4 Examples

We embark on the technical material in Section 5, but first we pause to examine some motivating examples typed in our proposed system.

Figure 2 gives the usage-polymorphic typings inferred by our system for a number of standard Haskell library functions.[3] When typing such library functions, we clearly must choose types that permit all possible uses; that is, that make no assumptions about how often the function or its partial applications are called. The types in the figure make use of usage polymorphism; without this, all variable annotations in the figure would take the "don't know" value ω, thus yielding dreadful results from the analysis.

- Usage polymorphism is used to describe dependencies between argument and result. The simplest possible example of this is the identity function, *id*, which simply returns its argument untouched. Clearly, if *id* is passed a use-once (use-many) thunk, it returns a use-once (use-many) thunk; this is expressed by its type.
- The short-circuit "and" is defined like this: $and\ a\ b = \mathsf{case}\ a\ \mathsf{of}\ \{\mathsf{True} \to b;\ \mathsf{False} \to \mathsf{False}\}$. Its type contains the same partial application dependency as Section 3.3's g_4 (in u_1). However, it also contains a dependency (in u_2) between its second argument and result: if the first argument is True the function behaves like *id*. This dependency does not conflict with the False case because the returned value (False) shared between all invocations of *and* is given the type Bool^{ω}, which is a subtype of Bool^{u_2}.
- The three-argument curried addition function *plus3* demonstrates that curried functions of more than two arguments still only have a single usage argument dealing with partial application. A type of the form $\cdot^{u_1} \to (\cdot^{u_2} \to (\cdot^{1} \to \cdot^{\omega})^{u_4})^{u_3}$ would have the constraint $u_1 \leq u_3$ as before (the first argument is used at least as many times as the first partial application), along with the unsurprising $u_2 \leq u_4$ (the second argument is used at least as many times as the second partial application), but in addition there is a constraint $u_1 \leq u_4$ (the first argument is also used as many times as the *second* partial application). These three constraints are resolved by unifying all four usage variables.
- The *flip* function is defined thus: $flip\ f\ x\ y = f\ y\ x$. Notice the "1" in its type, indicating that when it calls f it always fully applies it.
- Function composition *compose* is defined conventionally:

  $$compose\ f\ g\ x = f\ (g\ x)$$

  It exhibits a common pattern in which to each $\forall\alpha$ corresponds a $\forall u$, and all occurrences of α are decorated by that u. This pattern does not however justify abstracting over σ-types (*i.e.*, allowing *id* to have type $(\forall\alpha\ .\ \alpha \to \alpha)^{\omega}$), as we would then be unable to express useful types such as α^{1}: see [WPJ99, §6.2].

[3] Most of these should be self-explanatory. *apply*, written as infix `$` in Haskell, is strictly unnecessary but useful (because of its precedence) for avoiding excess parentheses in expressions. *build* and *augment* are used in 'cheap deforestation,' described in [Gil96, §3.4.2].

Figure 2 Some sample typings.

plus	::	$(\forall u_1 \,.$		$\mathsf{Int}^{u_1} \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^{u_1}$	$)^\omega$
id	::	$(\forall u_1 \,.$	$\forall \alpha \,.$	$\alpha^{u_1} \to \alpha^{u_1}$	$)^\omega$
and	::	$(\forall u_1, u_2 \,.$		$\mathsf{Bool}^{u_1} \to (\mathsf{Bool}^{u_2} \to \mathsf{Bool}^{u_2})^{u_1}$	$)^\omega$
apply	::	$(\forall u_1, u_2, u_3 \,.$	$\forall \alpha, \beta \,.$	$(\alpha^{u_1} \to \beta^{u_2})^{u_3} \to (\alpha^{u_1} \to \beta^{u_2})^{u_3}$	$)^\omega$
plus3	::	$(\forall u_1 \,.$		$\mathsf{Int}^{u_1} \to (\mathsf{Int}^{u_1} \to (\mathsf{Int}^1 \to \mathsf{Int}^\omega)^{u_1})^{u_1}$	$)^\omega$
flip	::	$(\forall u_1, u_2, u_3 \,.$	$\forall \alpha, \beta, \gamma \,.$	$(\alpha^{u_1} \to (\beta^{u_2} \to \gamma^{u_3})^1)^{u_2} \to (\beta^{u_2} \to (\alpha^{u_1} \to \gamma^{u_3})^{u_2})^{u_2}$	$)^\omega$
compose	::	$(\forall u_1, u_2, u_3, u_4 \,.$	$\forall \alpha, \beta, \gamma \,.$	$(\beta^{u_1} \to \gamma^{u_2})^{u_3} \to ((\alpha^{u_4} \to \beta^{u_1})^{u_3} \to (\alpha^{u_4} \to \gamma^{u_2})^{u_3})^{u_3}$	$)^\omega$
map	::	$(\forall u_1, u_2 \,.$	$\forall \alpha, \beta \,.$	$(\alpha^{u_1} \to \beta^{u_2})^\omega \to ((\mathsf{List}\ \alpha)^{u_1} \to (\mathsf{List}\ \beta)^{u_2})^\omega$	$)^\omega$
foldr	::	$(\forall u_1, u_2 \,.$	$\forall \alpha, \beta \,.$	$(\alpha^{u_1} \to (\beta^{u_2} \to \beta^{u_2})^1)^\omega \to (\beta^\omega \to ((\mathsf{List}\ \alpha)^{u_1} \to \beta^{u_2})^\omega)^\omega$	$)^\omega$
build	::	$(\forall u_1, u_2 \,.$	$\forall \alpha \,.$	$\Big(\forall \beta \,.\, (\alpha^{u_1} \to (\beta^{u_1} \to \beta^{u_1})^{u_1})^\omega \to (\beta^\omega \to \beta^{u_2})^1\Big)^1 \to (\mathsf{List}\ \alpha)^{u_2}$	$)^\omega$
augment	::	$(\forall u_1, u_2, u_3 \,.$	$\forall \alpha \,.$	$\Big(\forall \beta \,.\, (\alpha^{u_1} \to (\beta^{u_1} \to \beta^{u_1})^{u_1})^\omega \to (\beta^{u_2} \to \beta^{u_3})^1\Big)^\omega \to ((\mathsf{List}\ \alpha)^{u_2} \to (\mathsf{List}\ \alpha)^{u_3})^\omega$	$)^\omega$
zipWith	::	$(\forall u_1, u_2, u_3 \,.$	$\forall \alpha, \beta, \gamma \,.$	$(\alpha^{u_1} \to (\beta^{u_2} \to \gamma^{u_3})^1)^\omega \to ((\mathsf{List}\ \alpha)^{u_1} \to ((\mathsf{List}\ \beta)^{u_2} \to (\mathsf{List}\ \gamma)^{u_3})^{u_1})^\omega$	$)^\omega$
mkPair	::	$(\forall u_1 \,.$	$\forall \alpha, \beta \,.$	$\alpha^{u_1} \to (\beta^{u_1} \to (\mathsf{Pair}\ \alpha\ \beta)^{u_1})^{u_1}$	$)^\omega$
fst	::	$(\forall u_1 \,.$	$\forall \alpha, \beta \,.$	$(\mathsf{Pair}\ \alpha\ \beta)^{u_1} \to \alpha^{u_1}$	$)^\omega$

- Constructor functions like *mkPair* and destructor functions like *fst* can now be given regular types; in our earlier work [WPJ99] these required special typing rules.

5 Type system

Our goal is to construct an analysis which will take unannotated source terms and translate them into usage-annotated target terms, by means of type inference. Firstly, however, we must define the source and target languages, and present the rules that define when a target term is well-annotated.

Our previous paper [WPJ99] focussed on decent handling of polymorphism and of constructors and case over user-defined algebraic data types. Therefore the present paper makes the following major simplifications, for both source and target languages:

- We omit the Girard–Reynolds-style explicit type abstraction and application.
- We omit constructors and case, replacing them instead by the choice operator $e_1 \,[]\, e_2$. This selects one branch to evaluate, nondeterministically and thus in a manner not predictable by the type system.
- We restrict letrecs to a single binding rather than a mutually recursive group.

The language we deal with here exhausts all the technical complexity of the full language, with minimal clutter. Extending it to the full language is straightforward: all of our previous work on these features carries over directly into the present system.

The source language we deal with is defined in Figure 3. It is essentially a simplified, monomorphic, explicitly-typed variant of Core (see Section 3.1). The target language is defined below it, and adds usage annotations, along with usage polymorphism in the form of explict usage abstraction (generalisation) $\Lambda u \,.\, e$ and application (specialisation) $e\ \kappa$. Usage annotations κ may be constants 1 (used at most once) or ω (possibly used many times), or variables u, v, w. Annotations appear on top of types, and recursively on both sides of the function arrow. In terms, annotations appear as part of types, and also explicitly on lambdas and in usage applications. The lambda annotation denotes the use of the lambda abstraction, which is distinct from the use of its argument; in the full system constructors also take an annotation.

Figure 3 The source and target languages.

$$
\begin{array}{llcll}
\text{Source terms} & M & ::= & x & \mid\ n \\
 & & \mid & \lambda x : t \,.\, M & \mid\ M_1\ M_2 \\
 & & \mid & M_1 + M_2 & \mid\ M_1 \,[]\, M_2 \\
 & & \mid & \mathsf{letrec}\ x_1 : t_1 = M_1\ \mathsf{in}\ M & \\
\text{Source } t\text{-types} & t & ::= & t_1 \to t_2 & \mid\ \mathsf{Int}
\end{array}
$$

$$
\begin{array}{llcll}
\text{Target terms} & e & ::= & x & \mid\ n \\
 & & \mid & \lambda^\kappa x : \sigma \,.\, e & \mid\ e_1\ e_2 \\
 & & \mid & \Lambda u \,.\, e & \mid\ e\ \kappa \\
 & & \mid & e_1 + e_2 & \mid\ e_1 \,[]\, e_2 \\
 & & \mid & \mathsf{letrec}\ x_1 : \sigma_1 = e_1\ \mathsf{in}\ e & \\
\text{Target } \tau\text{-types} & \tau & ::= & \sigma_1 \to \sigma_2 & \mid\ \mathsf{Int} \\
 & & \mid & \forall u \,.\, \tau & \\
\text{Target } \sigma\text{-types} & \sigma & ::= & \tau^\kappa & \\
\text{Usage annotations} & \kappa & ::= & 1 \ \mid\ \omega \ \mid\ u, v, w &
\end{array}
$$

Figure 4 The subtype ($\preccurlyeq$) and primitive ($\leq$) orderings.

$$
\frac{\kappa_1 \leq \kappa_2 \quad \tau_1 \preccurlyeq \tau_2}{{\tau_1}^{\kappa_1} \preccurlyeq {\tau_2}^{\kappa_2}}\ (\preccurlyeq\text{-}\textsc{Annot})
\qquad
\frac{\tau_1 \preccurlyeq \tau_2}{\forall u \,.\, \tau_1 \preccurlyeq \forall u \,.\, \tau_2}\ (\preccurlyeq\text{-}\textsc{All-U})
$$

$$
\frac{}{\mathsf{Int} \preccurlyeq \mathsf{Int}}\ (\preccurlyeq\text{-}\textsc{Int})
\qquad
\frac{\sigma_3 \preccurlyeq \sigma_1 \qquad \sigma_2 \preccurlyeq \sigma_4}{\sigma_1 \to \sigma_2 \preccurlyeq \sigma_3 \to \sigma_4}\ (\preccurlyeq\text{-}\textsc{Arrow})
$$

$$
\frac{}{\omega \leq \kappa}
\qquad
\frac{}{\kappa \leq 1}
\qquad
\frac{}{u \leq u}
$$

5.1 Type rules

The type rules in Figure 5 define what it means for a target term to be well-typed. The judgement $\Gamma \vdash e : \sigma$ states that in context Γ, the target expression e can be given type σ.

Our rules are based on those of [WPJ99], and share with that system two key features. The system is (loosely) an affine linear one,[4] but using the syntactic occurrence function $occur(\cdot,\cdot)$[5] rather than the equivalent weakening and contraction rules; and has subsumption. Figure 4 defines the primitive ordering $\leq$ on usages and the induced structural subtype ordering $\preccurlyeq$ on types. Note that since the annotation 1 is more informative than ω, we have[6] $\omega \leq 1$ and $\tau^{\omega} \preccurlyeq \tau^{1}$.

The well-typing rules should be unsurprising. We use $|\sigma|$ to denote the topmost annotation of a type σ. Usage polymorphism is added by the rules ($\vdash$-UAbs) and ($\vdash$-UApp). Abstracted usage variables must not be free in the context, or be the topmost annotation of a type. This syntactic restriction—$\forall u\ .\ \cdot$ is defined to be a τ-type rather than a σ-type—is required to ensure that $|\sigma|$ is always well-defined (consider the possible meaning of $|\forall u\ .\ \tau^{u}|$); the same restriction is made in [GS00].

That this type system is sound with respect to an appropriate operational semantics may be demonstrated by a simple extension of the proofs in [WPJ99], or as a corollary of the proofs for a constrained polymorphic system such as [GS00].

6 Inference

We now present our inference algorithm, the principal contribution of this paper. We seek an algorithm that:

- Translates each source term to a target term whose erasure is the original source term.
- Yields a well-typed target term for any well-typed source term.
- Yields annotations that are as precise as possible.

The first point ensures we merely add information, and do not alter the semantics of the program. The second point ensures our analysis is "soft" [CF91]: our analysis does not affect the set of programs one is allowed to write. This permits it to be hidden from the programmer. The third point is a claim of optimality: we would like to generate the 'best possible' annotations.

6.1 Basic algorithm

The well-typing rules of Figure 5 specify what it means for a program to be type-correct, but do not in themselves constitute an algorithm for inferring type annotations for an unannotated source program. Such an algorithm must be constructed separately, and then proven sound with respect to the well-typing rules. Our inference algorithm is presented in Figures 6 and 7. The relation

$$C;\ \Gamma \blacktriangleright M \leadsto e : \sigma$$

may be read "In context Γ, source term M may be translated to term e which has type σ, under constraints C." The definition can be seen as an algorithm, with $\blacktriangleright$ a function from (Γ, M) to (e, σ, C). The algorithm closely follows standard ML type inference [MTHM97] except that we also gather constraints as we proceed.

Some notation: a constraint set is a set of atomic constraints $\kappa \leq \kappa'$, where κ, κ' are usage annotations (see Figure 3). The conjunction (union) of two constraint sets C_1, C_2 is written $C_1 \wedge C_2$. We write $\{\kappa = \kappa'\}$ to denote $\{\kappa \leq \kappa', \kappa' \leq \kappa\}$ and we write $\{P \Rightarrow C\}$ to denote the constraint set that is either C if P holds, or the trivial constraint set $\emptyset$ otherwise (note that this is a simple contraint, *not* an implicational constraint: P is constant). We write $\{\sigma_1' \preccurlyeq \sigma_1\}$ to denote the least set of atomic constraints $\kappa \leq \kappa'$ from which $\sigma' \preccurlyeq \sigma$ is derivable by the rules given in Figure 4.[7]

The $FreshAnnot(\cdot)$ operator in ($\blacktriangleright$-Abs) annotates a source type, yielding a target type with fresh usage variables everywhere. This is sufficiently general, since our subtyping is purely structural. The $FreshLUB$ operator in ($\blacktriangleright$-Choice) returns a constraint set and a type (possibly containing fresh usage variable annotations) such that under the constraints, the result type is the least upper bound of the arguments.

Once inference is complete we solve the constraint set, obtaining a substitution of usage annotations for usage variables that satisfies the constraints in the set (if the constraint set is insoluble, translation fails; Section 6.4 argues that this never occurs). When there are multiple such substitutions, we choose the *best:* the greatest under the obvious pointwise ordering on substitutions.[8]

There are two places where we explicitly deal with constraint sets: at the end, where we solve the constraint set; and at generalisation time, *i.e.*, in ($\blacktriangleright$-LetRec). Here a special *closure operator* is invoked.

6.2 Closure algorithm

The well-typing rules of Figure 5 give us no guidance in choosing where to usage-generalise a term, *i.e.*, where to insert usage abstractions in the translation. We choose simply to follow the Hindley–Milner approach, and make all letrecs usage-polymorphic (Figure 7). This is done by applying a *closure operator* to the constraint set generated by each binding group, to compute the most general usage types for the bindings it contains.

This closure operator $Clos(\cdot,\cdot,\cdot)$ is the core of the inference, and the major technical contribution of this paper. The usage-monomorphic system presented in [WPJ99] may be obtained by putting $Clos(C, \Gamma, \overline{\tau_i{}^{\kappa_i}}) = (C, [\,], [\,])$; instead, we present below our new algorithm for generating unconstrained usage-polymorphic types by means of approximation.

Note that our target language can express general impredicative ("nested") polymorphism; our algorithm infers only

[4] *i.e.*, a linear type system [Gir87] admitting weakening but not contraction [Jac94].

[5] The value of $occur(x, e)$ is the number of syntactic occurrences of the variable x in the term e, except that for choice terms $e_1 \mathbin{[\!]} e_2$ it is $\mathbf{max}(occur(x, e_1), occur(x, e_2))$.

[6] This is a notational change relative to [WPJ99].

[7] As an aside, note that this means that expressions such as $\{\mathsf{Int}^{u_1} \preccurlyeq (\mathsf{Int}^{u_2} \to \mathsf{Int}^{u_3})^{u_4}\}$ are undefined.

[8] The algorithm of [WPJ99, §7.2] is used to find this solution, which exists and is unique due to the simple nature of the constraints.

Figure 5 Well-typing rules for the target language.

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (\vdash\text{-Var}) \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}^\omega}\ (\vdash\text{-Int}) \qquad \frac{\begin{array}{c}\Gamma, x_1 : \sigma_1 \vdash e_1 : \sigma_1 \qquad \Gamma, x_1 : \sigma_1 \vdash e : \sigma \\ (\mathit{occur}(x_1, e) + \mathit{occur}(x_1, e_1)) > 1 \Rightarrow |\sigma_1| = \omega\end{array}}{\Gamma \vdash \mathsf{letrec}\ x_1 : \sigma_1 = e_1\ \mathsf{in}\ e : \sigma}\ (\vdash\text{-LetRec})$$

$$\frac{\Gamma \vdash e_i : \sigma \qquad i = 1, 2}{\Gamma \vdash e_1 \,[]\, e_2 : \sigma}\ (\vdash\text{-Choice}) \qquad \frac{\Gamma \vdash e_i : \mathsf{Int}^1 \qquad i = 1, 2}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}^\omega}\ (\vdash\text{-PrimOp})$$

$$\frac{\begin{array}{c}\Gamma, x : \sigma_1 \vdash e : \sigma_2 \\ \mathit{occur}(x, e) > 1 \Rightarrow |\sigma_1| = \omega \\ \mathit{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq \kappa \quad \text{for all } y \in \Gamma\end{array}}{\Gamma \vdash \lambda^\kappa x : \sigma_1 \,.\, e : (\sigma_1 \to \sigma_2)^\kappa}\ (\vdash\text{-Abs}) \qquad \frac{\Gamma \vdash e_1 : (\sigma_1 \to \sigma_2)^1 \qquad \Gamma \vdash e_2 : \sigma_1}{\Gamma \vdash e_1\ e_2 : \sigma_2}\ (\vdash\text{-App})$$

$$\frac{\Gamma \vdash e : \tau^\kappa \qquad \overline{u_i} \notin \mathit{fuv}(\Gamma) \cup \mathit{fuv}(\kappa)}{\Gamma \vdash \Lambda \overline{u_i} \,.\, e : (\forall \overline{u_i} \,.\, \tau)^\kappa}\ (\vdash\text{-UAbs}) \qquad \frac{\Gamma \vdash e : (\forall \overline{u_i} \,.\, \tau)^\kappa}{\Gamma \vdash e\ \overline{\kappa_i} : (\tau[\overline{u_i := \kappa_i}])^\kappa}\ (\vdash\text{-UApp}) \qquad \frac{\Gamma \vdash e : \sigma' \qquad \sigma' \preccurlyeq \sigma}{\Gamma \vdash e : \sigma}\ (\vdash\text{-Sub})$$

Figure 6 Basic type inference rules (omitting ($\blacktriangleright$-LetRec)).

$$\frac{\text{fresh } \overline{v_i} \qquad \tau \text{ a usage-monotype}}{\emptyset;\ \Gamma, x : (\forall \overline{u_i} \,.\, \tau)^\kappa \blacktriangleright x \rightsquigarrow x\ \overline{v_i} : (\tau[\overline{u_i := v_i}])^\kappa}\ (\blacktriangleright\text{-Var}) \qquad \frac{}{\emptyset;\ \Gamma \blacktriangleright n \rightsquigarrow n : \mathsf{Int}^\omega}\ (\blacktriangleright\text{-Int})$$

$$\frac{\begin{array}{c}C_i;\ \Gamma \blacktriangleright M_i \rightsquigarrow e_i : \sigma_i \qquad i = 1, 2 \\ (C_3, \sigma) = \mathit{FreshLUB}(\sigma_1, \sigma_2)\end{array}}{C_1 \wedge C_2 \wedge C_3;\ \Gamma \blacktriangleright M_1 \,[]\, M_2 \rightsquigarrow e_1 \,[]\, e_2 : \sigma}\ (\blacktriangleright\text{-Choice}) \qquad \frac{C_i;\ \Gamma \blacktriangleright M_i \rightsquigarrow e_i : \mathsf{Int}^{\kappa_i} \qquad i = 1, 2}{C_1 \wedge C_2;\ \Gamma \blacktriangleright M_1 + M_2 \rightsquigarrow e_1 + e_2 : \mathsf{Int}^\omega}\ (\blacktriangleright\text{-PrimOp})$$

$$\frac{\begin{array}{c}\text{fresh } v \qquad \sigma_1 = \mathit{FreshAnnot}(t_1) \qquad C_1;\ \Gamma, x : \sigma_1 \blacktriangleright M \rightsquigarrow e : \sigma_2 \\ C_2 = \{\mathit{occur}(x, e) > 1 \Rightarrow |\sigma_1| = \omega\} \qquad C_3 = \bigwedge_{y \in \Gamma} \{\mathit{occur}(y, e) > 0 \Rightarrow |\Gamma(y)| \leq v\}\end{array}}{C_1 \wedge C_2 \wedge C_3;\ \Gamma \blacktriangleright \lambda x : t_1 \,.\, M \rightsquigarrow \lambda^v x : \sigma_1 \,.\, e : (\sigma_1 \to \sigma_2)^v}\ (\blacktriangleright\text{-Abs})$$

$$\frac{C_1;\ \Gamma \blacktriangleright M_1 \rightsquigarrow e_1 : (\sigma_1 \to \sigma_2)^\kappa \qquad C_2;\ \Gamma \blacktriangleright M_2 \rightsquigarrow e_2 : \sigma_1' \qquad C_3 = \{\sigma_1' \preccurlyeq \sigma_1\}}{C_1 \wedge C_2 \wedge C_3;\ \Gamma \blacktriangleright M_1\ M_2 \rightsquigarrow e_1\ e_2 : \sigma_2}\ (\blacktriangleright\text{-App})$$

predicative usage type "schemes". However, impredicative types may be present in the initial context or in explicit type signatures, and will then be handled correctly. Such types have been shown to be useful in other contexts (such as in the type of `runST` [LPJ94]) and might conceivably prove to be so here, but inference is known to be undecidable in this case. A possible approach suggests itself from the common pattern observed for *compose* in Section 4: we could attempt to follow each nested type quantifier with a usage quantifier. Such an approach might well improve the type of functions such as *build* (Figure 2). We have not tried this.

Figure 8 shows the closure algorithm. The input is a triple: a constraint set C for the binding group being considered, a context Γ in which the generalisation is to be done, and a vector of the types (usage-monomorphic by construction) and topmost annotations $\overline{\tau_i{}^{\kappa_i}}$ of the bindings. The output is a reduced constraint set, a vector of usage variables over which to generalise, and a unifying substitution to be applied.

The closure algorithm is necessarily *approximate*—in general, simple usage polymorphism does not admit most-general types, and a term such as *plus3* may have multiple incomparable types between which we choose essentially arbitrarily (Section 6.3). The algorithm does however guarantee to yield *maximally-applicable* types—types for binders which do not restrict their use. This is necessary to satisfy the second requirement of Section 6.

We take a moment to introduce some essential notation.

For target types, we define a *positive occurrence* of a usage variable u in a type τ or σ to be one annotating a covariant position, and a *negative occurrence* to be one annotating a contravariant position. In the case of a σ-type, the topmost annotation is considered to be covariant. We define functions $\mathit{fuv}^+(\sigma)$ ($\mathit{fuv}^-(\sigma)$) as the set of usage variables occurring positively (negatively) in σ; similarly for τ-types. Thus if $\sigma = ((\alpha^{u_1} \to \beta^{u_2})^{u_3} \to (\gamma^{u_4} \to \delta^{u_5})^{u_6})^{u_7}$, we have $\mathit{fuv}^+(\sigma) = \{u_1, u_5, u_6, u_7\}$ and $\mathit{fuv}^-(\sigma) = \{u_2, u_3, u_4\}$. The notation $\mathit{fuv}(\sigma)$ denotes the set of usage variables occurring positively *or* negatively in σ; *i.e.*, $\mathit{fuv}^+(\sigma) \cup \mathit{fuv}^-(\sigma)$. We use ε to range over $\{+, -\}$. The relation $\leq_C$ is the partial

Figure 7 Type inference rule ($\blacktriangleright$-LetRec).

$$\tau_1{}^{v_1} = \mathit{FreshAnnot}(t_1)$$

$$C_1;\ \Gamma, x_1 : \tau_1{}^{v_1} \blacktriangleright M_1 \rightsquigarrow e_1 : \sigma_1'$$

$$C_2 = \{\sigma_1' \preccurlyeq \tau_1{}^{v_1}\}$$

$$(C_3, \overline{u_k}, S) = \mathit{Clos}(C_1 \wedge C_2, \Gamma, \tau_1{}^{v_1})$$

$$C_4;\ \Gamma, x_1 : (\forall \overline{u_k}\ .\ S\tau_j)^{v_1} \blacktriangleright M \rightsquigarrow e : \sigma$$

$$C_5 = \{(\mathit{occur}(x_1, e) + \mathit{occur}(x_1, e_1)) > 1 \Rightarrow v_1 = \omega\}$$

$$C_3 \wedge C_4 \wedge C_5;\ \Gamma$$

$$\blacktriangleright\quad \mathsf{letrec}\ x_1 : t_1 = M_1\ \mathsf{in}\ M$$

$$\rightsquigarrow\quad \mathsf{letrec}\ x_1 : (\forall \overline{u_k}\ .\ S\tau_1)^{v_1} = \Lambda \overline{u_k}\ .\ Se_1[x_1 := x_1\ \overline{u_k}]\ \mathsf{in}\ e : \sigma$$

Figure 8 The closure operation.

$$\mathit{Clos}(C_0, \Gamma_0, \overline{\tau_{0i}{}^{\kappa_{0i}}}) = (C', \overline{u_i}, S) \quad \text{where}$$

$$\begin{aligned}
(C, S_0) &= \mathit{TransitiveClosure}(C_0) \\
\Gamma &= S_0\Gamma_0 \\
\overline{\tau_i{}^{\kappa_i}} &= S_0\overline{\tau_{0i}{}^{\kappa_{0i}}} \\[1ex]
F &= \{u^\varepsilon \mid u \in \mathit{fuv}^\varepsilon(\overline{\tau_i})\} \\
X &= \mathit{fuv}(\Gamma) \cup \mathit{fuv}(\overline{\kappa_i}) \\[1ex]
\Phi(A) &= A \cap F \\
&\quad \cap \{u^\varepsilon \mid u^\varepsilon \in A,\ \neg\exists x \in X\ .\ x \leq_C^\varepsilon u\} \\
&\quad \cap \{u^-,\ v^+ \mid u \leq_C v,\ u^- \in A,\ v^+ \in A\} \\
G &= \mathit{gfp}(\Phi) \\[1ex]
(\sim) &= \{(u,v),\ (v,u) \mid u \leq_C v,\ u^- \in G, v^+ \in G\} \\
\mathcal{U} &= \{u \mid u^\varepsilon \in G\}\ /\ (\sim) \\
\overline{u_i} &= \text{a vector containing one representative} \\
&\quad\ \text{from each equivalence class in } \mathcal{U}
\end{aligned}$$

$$\begin{aligned}
S &= \{(x \mapsto u_i) \mid \exists u^-, v^+ \in G\ .\ u \leq_C x,\ x \leq_C v,\ u_i \in [u]_{(\sim)}\} \\
C' &= SC \wedge \{x \leq \omega \mid \neg\exists u^- \in G\ .\ u \leq_C x,\ \exists v^+ \in G\ .\ x \leq_C v\} \\
&\quad \wedge \{1 \leq x \mid \exists u^- \in G\ .\ u \leq_C x,\ \neg\exists v^+ \in G\ .\ x \leq_C v\}
\end{aligned}$$

order induced by constraint set C, and we write $u \leq_C^+ v$ for $u \leq_C v$ and $u \leq_C^- v$ for $v \leq_C u$.

The intent of the closure operation is to compute the largest possible set of variables to generalise over. There are four reasons a variable should not be generalised: (i) it may be constrained already such that its range is restricted; (ii) it may be possible that it will become so constrained later in the inference process; or (iii) it may be a topmost annotation and thus syntactically ungeneralisable.[9]

Further, a variable should not be generalised if (iv) it does not express a dependency. When inferring the type of an expression it is safe to return a smaller (more informative) type, since subsumption will be applied to the result type. We therefore desire to infer the smallest possible type for the expression. This means that positive variables should be minimised, and negative variables maximised. A *dependency* occurs when a positive variable is constrained to be greater than a negative variable; in this case the monomorphic solutions are incomparable, and instead we unify the variables and generalise (as discussed in Section 3.3). Condition (iv) states (due to subsumption) that we need not generalise other variables.

The closure operation first implements condition (i) by substituting values for all completely constrained variables.[10] These values are obtained by means of an auxiliary partial constraint solver *TransitiveClosure* which forms the transitive closure of the constraints collected in C_0. If this succeeds it returns a set of substitutions S for variables whose values are completely determined by C_0, and a residual constraint set C constraining these variables directly to their values, and the remaining variables equivalently to C_0. If C_0 is unsatisfiable, *TransitiveClosure* and the entire translation fail; Section 6.4 argues that this never occurs.

The set F contains the free usage variables of the result types (which are candidates for generalisation), tagged by the polarity of their occurrence,[11] and X contains free usage variables of the context (for implementing condition (ii)) and of the topmost annotations (for implementing condition (iii)). Variables appearing in or constrained by other variables in X should not be generalised. Given these, G is the largest subset ('greatest fixed point', hence *gfp*) of F such that conditions (ii) through (iv) hold: the third conjunct requires that variables in G not be constrained by variables in X,[12] and the fourth conjunct requires that all positive (negative) variables be greater (less) than a negative (positive) variable.

Having found the set of variables to be generalised, we now perform the required unifications. Note that the constraint set C may contain many variables that do not occur in the types $\overline{\tau_i}$, but have been generated internally during inference. If two variables u and v are to be unified and generalised, then all variables in between them must be so also. In general, we have a number of 'clusters' of related variables in F, constraining other variables between them. One such cluster might look as follows (this is an outline of a Hasse diagram, where apart from the typical variable x all variables

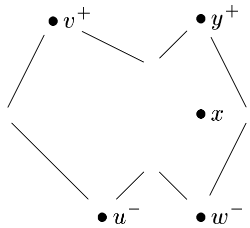
[9] This syntactic restriction is discussed in Section 5.1.

[10] In our simple two-point lattice, a variable's range is either unrestricted or restricted to a single value; this is likely not to be the case for the system discussed below in Section 9, and so this part of the algorithm will need to be rewritten.

[11] If a usage variable u were to appear both positively and negatively in the result types, it would appear twice in F, once as u^- and once as u^+. It can be shown, however, that with the translation algorithm as given this never occurs.

[12] We need only check that positive (negative) variables are not greater (less) than variables in X; the other direction is ensured by the fourth conjunct, by transitivity of $\leq$.

in the central region have been omitted for clarity):



Here u and w are negative, v and y are positive, and $u \le v$, $w \le y$, $u \le y$, $w \le v$. Clearly u, v, w, y must be unified, but so too must x and all variables like it, in the region defined by $\{x \mid (u \le x \vee w \le x) \wedge (x \le v \vee x \le y)\}$. Furthermore, variables above (below) such a unified and generalised group must be guaranteed to lie above (below) any instance of it. This we achieve by constraining such variables to 1 (ω).

The unification is performed as follows: we first construct the equivalence relation ($\sim$) which relates pairs of dependent variables in G. From this we construct equivalence classes of variables in G, place an arbitrary ordering on these classes, and pick an arbitrary representative element from each. The result is a vector of variables $\overline{u_i}$. We then create the unifying substitution: if a variable lies inside a cluster's region, we replace it with the representative variable from the appropriate equivalence class.

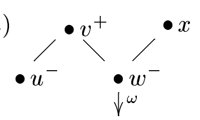
Finally, we return the now-unified constraint set,[13] augmented with constraints that force variables lying above (below) a cluster to 1 (ω). Other variables are untouched. We also return the vector of unified variables over which to generalise, and the substitution to be applied.[14]

6.3 Optimisations and design choices

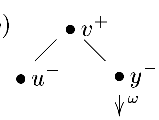
During the implementation, two possible optimisations suggested themselves. Variables local to C' that are completely unconstrained may be set to 1 immediately, anticipating the global maximisation performed at the end of the inference; and constraints added to C' may also be added as substitutions to S, and performed earlier.

Furthermore, consider the following situations:

(a)



(b)



Here u, v, w are in F, and x, y are in X. In (a), according to the algorithm none of the variables may be generalised, because they are all related to x, a variable appearing in the context. But in fact we *can* generalise u and v, at the cost of forcing w to ω, as indicated by the arrow.

In fact, x need not be in X for this case to be interesting. Consider that the same constraints arise when typing *plus3*, as discussed in Section 4. Now if we knew at closure time that *plus3* was likely to be very often partially applied to a single argument, we could force x to ω, and get the (in this context) more useful type $(\forall u\,.\,\cdot^{\omega} \to (\cdot^{u} \to (\cdot^{1} \to \cdot^{\omega})^{u})^{\omega})^{\omega}$, rather than $(\forall u\,.\,\cdot^{u} \to (\cdot^{u} \to (\cdot^{1} \to \cdot^{\omega})^{u})^{u})^{\omega}$.

In (b), we can generalise u and v at the cost of forcing y, a variable in the *context*. This is certainly not HM-style behaviour: we expect the context to affect bindings inferred within it, not the other way around; but we know it is always safe to force such a variable to ω (forcing to 1 may cause inference subsequently to fail).

There is no right choice for whether or not to force such variables, because the types resulting are incomparable (consider the two types for *plus3* above). In some cases we certainly should not force: if we are still unable to generalise after forcing, we have lost information to no purpose. In the absence of any general rule, we choose simply to ignore these possibilities, and follow the algorithm as given.[15]

6.4 Soundness and precision

In Section 6 we gave three requirements for our inference algorithm. Our algorithm certainly satisfies the first two; maximum precision turns out to be something of a movable feast.

It is clear by inspection of the rules that the erasure of the target term resulting from the translation is identical to the original source term. Thus our algorithm does not alter the meaning of the program.

Soundness is slightly less trivial. We require that translations be well-typed whenever source terms are; *i.e.*, if $\vdash M : t$ then $C;\ - \blacktriangleright M \rightsquigarrow e : \sigma$, where C is satisfiable and for all substitutions S satisfying C we have $\vdash Se : S\sigma$.

The proof of this makes use of a lemma: if $C_i;\ \Gamma \blacktriangleright M_i \rightsquigarrow e_i : \sigma_i$ and all the C_i are satisfiable, then the conjunction $\bigwedge_i C_i$ is also satisfiable. We proceed by structural induction on the inference derivation tree. All the rules in Figure 6 can be seen by inspection to preserve the well-typedness property and the lemma, since usage variables are only ever constrained to ω or a fresh usage variable. This leaves ($\blacktriangleright$-LetRec). The proof for this requires us to show that the unifying substitution and additional constraints generated by the closure operation of Figure 8 do not violate the lemma. While we have not yet conducted a formal proof of this result, we are confident that it holds.

The final requirement was that annotations are 'as precise as possible'. For our earlier usage-monomorphic system [WPJ99], this could be simply interpreted as 'inference yields target programs with the greatest possible number of 1 annotations'. The introduction of usage polymorphism significantly complicates the accuracy ordering. As we saw in Section 6.3, terms such as *plus3* have multiple incomparable usage-polymorphic types, and thus there is no maximally precise type for our algorithm to find. However, experience with the implementation leads us to believe our algorithm behaves in a reasonable manner, and we hope (see Section 9) to give a more formal treatment in a subsequent paper.

[13] This unification SC can in fact be implemented alternatively as $C \wedge \{x \le u, u \le x \mid (x \mapsto u) \in S\}$.

[14] Since variables occurring in the context cannot constrain generalised variables, the substitution is (luckily) the identity outside the binding group.

[15] Of course, the problem trivially disappears in the presence of constrained polymorphism because the constraint set is used unapproximated.

7 Implementation

We have implemented our analysis within the Glasgow Haskell Compiler (GHC) [PJS98]. GHC is an optimising compiler, and it operates by translating Haskell input into Core, performing a long series of optimisation passes over the Core program, and finally translating the resulting Core program into STG code and then object code. The Core language is typed, and types are maintained throughout the optimisation phase of the compiler.

In our implementation, as we have seen, usage information is made an integral part of the type language. This means that we can ensure usage annotations are preserved through all the various transformations performed, and at any point we can perform a type-check to verify that they are correct (*i.e.*, that no optimisation pass has unintentionally violated the usage annotations). This was not entirely straightforward to achieve, and in fact one transformation, common subexpression elimination, yields code that may require a full re-inference pass to preserve well-typedness. For most transformations, however, careful local manipulation of annotations is sufficient.

Our 'plan of attack' is that of [WPJ99, §2.3]: we perform an initial usage type inference soon after translation into Core; subsequent transformations preserve usage type soundness, but at any point we may choose to perform another inference to improve the accuracy of the types (recall that a decidable inference of any interesting operational property is necessarily an approximation). Finally, we perform a final inference just prior to translation to STG, to ensure maximally-accurate usage information is available when deciding which code to generate for each thunk.

Usage information is used in two places. Firstly, it is used by the code generator to choose whether or not to generate an updatable thunk and push an update frame on entry. Secondly, it used by the optimisation passes. The major (currently the only) information channel here is the usage annotations on lambda expressions. Each lambda knows whether it is a 'one-shot lambda' (*i.e.*, a lambda that will be applied at most once) or not. As shown in [San95, PJPS96] and discussed in our earlier paper [WPJ99, §2], identifying such one-shot lambdas greatly increases the scope for code-floating transformations such as inlining and full laziness. Previously a couple of rather ugly hacks identified a few special cases where lambdas must be one-shot; our inference identifies all these, and many more.

7.1 Measurements

Table 2 shows the overall results of adding our analysis to the optimising compiler GHC. All standard libraries were compiled with the analysis, in addition to the program under test. For each program, the change in total bytes allocated and in run time is given relative to the version of the compiler and libraries without usage inference. We also list the percentage of thunks demanded at most once during execution that were statically identified as such by the analysis, both with optimisation on and off (optimisation was always turned on for the standard libraries).

It can be seen that both total allocations and run time decrease significantly relative to the compiler without the analysis. Note, however, that essentially *no* used-once thunks are identified in the code generated. We hypothesise that the optimiser is taking advantage of usage information provided it by our analysis to remove identified used-once thunks altogether. Unfortunately, a large number of unidentified used-once thunks remain.

Table 2 Measurements

Program	Allocs	Run time	1-thunks found	
	`-O`	`-O`	`-O`	`-Onot`
`cichelli`	-1.16%	-5.23%	0.00%	35.00%
`compress`	+1.12%	-1.94%	0.00%	0.00%
`event`	-3.34%	+0.26%	0.00%	0.00%
`exp3_8`	0.00%	-7.33%	0.00%	0.04%
`fibheaps`	-8.78%	-9.13%	0.00%	3.00%
`gamteb`	-7.28%	-16.44%	0.88%	17.00%
`queens`	0.00%	+5.88%	0.00%	71.00%
`solid`	0.00%	+6.16%	0.00%	0.00%
Average	-2.49%	-3.74%		

IMPORTANT NOTE: The figures in this table are known to be incorrect. Corrected figures will appear in the formal proceedings.

Our other results (not shown) show that the costs of the analysis are not high. The standard libaries (and hence binaries that link them in) increase by about 1% in size, although many modules decrease slightly; and compile times increase by only a small factor in most cases (although a few modules nearly double in time, presumably due to inefficiency in our constraint-solution algorithm).

7.2 Discussion

The improvements in total allocations and run time are quite encouraging. A 3% improvement is definitely significant given the number of other optimisations already performed by GHC.

However, even with our analysis working on both libraries and program, and optimisation turned off, a program like `compress` allocates 810,000 thunks of which 680,000 are entered at most once—but our analysis identifies precisely *one* of them! For a number of programs our analysis performs respectably, but for many the results are like those for `compress`. What is going wrong?

A more complete answer will have to await deeper analysis, but initial investigations suggest a likely culprit is our treatment of data structures (discussed in our previous paper [WPJ99]). We currently identify, for example, the usage of the spine of a list with the usage of its elements; this is bad for the common pattern of an intermediate data structure, where the list itself is used once but the contents may be used multiple times. We are presently investigating the impact of using a less drastic typing for data structures.

Another possible factor is our choice to restrict the analysis to monovariance: we generate just a single copy of each function, and provide that with as polymorphic a type as possible. Would a polyvariant analysis do better? As a very crude test[16] we told the code generator to treat thunks annotated with usage variables as used-once, rather than used-many: this effectively specialised all functions at 1, at the cost of soundness. Surprisingly, although some of the programs in our test suite aborted with the diagnostic "single-entry thunk re-entered", a large fraction of the programs in our test suite encountered no problems; yet the percentage

[16]Like Rutherford's experiment, this was in fact an accident!

of used-once thunks allocated went up dramatically. This indicates that specialisation would very likely improve the situation dramatically. Helpfully, the specialisation machinery is all currently present in GHC anyway, and adding the required specialisations would be relatively straightforward.

8 Related work

Constrained polymorphism was introduced independently by Curtis [Cur90] and by Aiken and Wimmers [AW93], and a significant body of work has developed since. Constrained polymorphism is closely related to *bounded* polymorphism, introduced by Cardelli *et al.* in [CW85, CMMS94]. With the addition of recursive types, bounded polymorphism becomes equivalent to constrained polymorphism [PS96], and this latter setting seems more natural for many problems. Trifonov and Smith's [TS96] is a key paper dealing with the theory of subtyping in this context.

Much work has also been done on the type inference problem for these systems [JP99]; in practice this reduces to the problem of constraint simplification [AWP97, Pot98, FF99, MW97, FM90]. Our subtype ordering is purely structural, making simplification significantly easier than the general case.

Constraint simplification attempts to find a minimal description of a given type. We go further, and restrict the form of the description in a way that forces it to be smaller, but also forces it (in general) to be *approximate*. There are two approximation algorithms of which we are aware. Cardelli's 'greedy algorithm' [Car93] (for a bounded polymorphic language) resolves all subtyping constraints immediately (*i.e.*, on generation at application sites), performing unification on type variables as necessary to ensure that constraints need never be propagated. Slightly less greedy is the algorithm of Nordlander [Nor98] (for a constrained polymorphic language), which resolves only *local* subtyping constraints immediately, and propagates constraints on variables in the environment further upwards. Nordlander's algorithm is relatively *ad hoc*, motivated by a desire for the subtype inference algorithm to follow the standard unification algorithm as closely as possible. Our closure algorithm provides a much more formal presentation of constraint approximation, and should be more amenable to discussion and analysis.

Flanagan and Felleisen [FF99] discuss in detail the problem of ranking multiple solutions to a constraint set so as to select the 'best' or most accurate one. Their accuracy ordering $\sqsubseteq_s$ is *covariant* on function types, and is distinct from their subtype ordering which is, as usual, contravariant on function types. This may provide a formal basis for our pointwise (*i.e.*, covariant) ordering of constraint-set solutions, described in [WPJ99] and retained in the present paper (Section 6.1).

The idea of usage polymorphism itself is in no way new. It is proposed in the paper that started us off, [TWM95], although in the context of either code specialisation or run-time usage-passing. A similar notion of annotation polymorphism is also familiar in the flow analysis community, under the term *polyvariance* [Bul84, JW95, WJ98]; many flow analyses are abstract interpretations rather than type-based analyses, but it has been shown that the two are closely related [Jen91]. Polyvariance has been applied to many different analyses, notably here binding-time analysis [DHM95], and so its application to usage is unsurprising.

Gustavsson [Gus98] infers both usage information (in exactly our sense) and update-marker-check information for a language similar to ours. Recently [GS00] he has added constrained polymorphism (the 'natural choice' we mentioned in Section 3.4) to his analysis, and believes that it will prove feasible in practice. Similarly, the Clean uniqueness typing system of Barendsen *et al.* [BS96] features constrained uniqueness polymorphism for lambda-lifted functions and data constructors. Our analysis takes a different approach; we believe that simple polymorphism will be cheaper and hope that it will be sufficient in practice. We do not attempt to address the update-marker-check problem.

Systems based on *soft typing* [CF91], in common with our analysis, must be able to type all valid programs; and in both cases maximum benefit occurs when the types inferred are as accurate as possible. However, soft typing is necessarily visible to the programmer: it is intended for debugging; in contrast, our analysis is intended to be invisible: it is intended to guide optimisation. Further, type mismatches in a soft typing system are typically resolved by arbitrary static coercions supported by dynamic checks; in our system, type mismatches provably do not occur, by construction.

Work relating to usage analysis but not specifically to usage polymorphism is discussed in our earlier paper, [WPJ99].

9 Conclusions and future work

We have presented a novel algorithm that infers simple polymorphic types (rather than the natural constrained polymorphic types) for a language with subtyping, by means of judicious constraint approximation. This algorithm should be applicable to any analysis that currently uses constrained polymorphism, such as the binding-time analysis of [DHM95]. We have implemented our analysis in the Glasgow Haskell Compiler, and performed some preliminary measurements of its effectiveness. These measurements are encouraging, although there is still a significant amount of work to be done in maximising the benefit to the compiler of our analysis.

There are a number of matters remaining to be investigated.

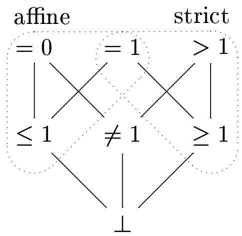
Clearly we need a deeper formal understanding of exactly which types and annotations are found by our algorithm. We have some ideas as to how this might be achieved. Our algorithm may be viewed as taking a type[17] and attempting to find an optimal description for it, given certain restrictions on the form of that description (namely, that it be only simply polymorphic). Clearly we must maintain soundness and so for closed types at least, the type we choose is (almost) the least type describable that is both greater than the given type and as applicable[18] as the given type, except that in certain cases there is no such least type. We defer a complete treatment of this to a subsequent paper discussing the theoretical issues underlying our algorithm.

One major target we have in view is the extension of our usage annotations from the simple two-point lattice $\omega \leq 1$ to the following seven-point 'lattice' (the apparently-missing

[17] We here identify a type σ with the set $[\![\sigma]\!]$ of its ground instances, and order types by reverse set inclusion: $\sigma \sqsubseteq \sigma'$ iff $[\![\sigma']\!] \subseteq [\![\sigma]\!]$.

[18] Applicability is a notion related to the pessimising translation performed for exported functions; the latter simply returns the least maximally-applicable type of the same shape as its input. Applicability requires covariant annotations to lie in the inverse primitive ordering, and ignores contravariant annotations.

top element would be meaningless). This subsumes absence, strictness, and linearity, in addition to the present (affine linear) usage analysis:



We term this the Bierman lattice, after [Bie92]; it is closely related to the usage intervals of [Ses91, §5] and to the logics of [Wri96]. We are close to completing the extension of our monomorphic analysis to this lattice, and believe that it should be possible to extend the usage-polymorphic inference algorithm we have presented similarly. We aim thereby to have the compiler perform all these analyses simultaneously.

There remains also significant implementation effort, convincing GHC to make the best possible use of the information afforded by the analysis, and ensuring that the usage types (which are surprisingly fragile) are preserved by the various transformations within it.

Acknowledgements

We wish to thank Luca Cardelli, Karl-Filip Faxén, Jörgen Gustavsson, Fritz Henglein, Mark Jones, Martin Müller, Martin Odersky, Andrew Pitts, François Pottier, Martin Sulzmann, Josef Sveningsson, Valery Trifonov, and the anonymous reviewers for helpful comments and fruitful discussions by email and in person. Their input has made this presentation much clearer.

References

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture (FPCA'93), Copenhagen, Denmark*, pages 31–41. ACM Press, 1993.

[AWP97] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping (extended abstract). In *Proceedings of TACS'97, International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan*, number 1281 in Lecture Notes in Computer Science, pages 47–76. Springer–Verlag, September 1997. Journal version appears as [AWP99].

[AWP99] Alexander Aiken, Edward L. Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping (extended abstract). *Higher-Order and Symbolic Computation*, 12(3), October 1999. Journal version of [AWP97].

[Bie92] Gavin Bierman. Type systems, linearity and functional languages. In *CLICS Workshop, Aarhus, March 23–27, 1992*, 1992. Appears as Technical Report DAIMI PB-397-I, Department of Computer Science, Aarhus University, May 1992, pp. 71–91.

[BS96] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[Bul84] M. A. Buluyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[Car93] Luca Cardelli. An implementation of $F_{<:}$. SRC Research Report 97, Digital Equipment Corporation, Systems Research Center, February 23 1993. Revised June 1997.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. *ACM SIGPLAN Notices*, 26(6):278–292, June 1991.

[CMMS94] Luca Cardelli, John C. Mitchell, Simone Martini, and Andre Scedrov. An extension of System F with subtyping. *Information and Computation*, 109(1/2):4–56, February 1994.

[Cur90] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Palo Alto Research Center, February 1990.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[DHM95] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Mycroft [Myc95]. Extended version, obtained from third author's home page.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In Mycroft [Myc95].

[Fax97] Karl-Filip Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, 1997. Appears as Research Report TRITA-IT R 97:08.

[FF99] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, March 1999.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 72(2):155–175, June 1990.

[Gil96] Andrew John Gill. *Cheap Deforestation for Non-Strict Functional Languages*. PhD thesis, University of Glasgow Department of Computing Science, January 1996.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[GS00] Jörgen Gustavsson and Josef Sveningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Implementation of Functional Languages 12th International Workshop (IFL'00)*, Aachen, Germany, September 2000. Proceedings to be published as a technical report of the Department of Computer Science, RWTH Aachen.

[Gus98] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In ICFP [ICF98].

[ICF98] *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, 1998.

[Jac94] Bart Jacobs. Semantics of weakening and contraction. *Annals of Pure and Applied Logic*, 69:73–106, 1994.

[Jen91] Thomas P. Jensen. Strictness analysis in logical form. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Boston, MA*, number 788 in Lecture Notes in Computer Science. Springer–Verlag, August 1991.

[JP99] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Available `http://www.cs.purdue.edu/homes/palsberg/publications.html`, June 1999.

[JW95] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In Mycroft [Myc95], pages 207–224.

[LPJ94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press. Extended version appears as [LPJ95].

[LPJ95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8:293–341, 1995.

[Mar93] Simon Marlow. Update avoidance analysis by abstract interpretation. In *Glasgow Workshop on Functional Programming, Ayr*, Springer Verlag Workshops in Computing Series, July 1993.

[Mos93] Christian Mossin. Polymorphic binding time analysis. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, July 21 1993.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML — Revised.* MIT Press, 1997.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *International Conference on Functional Programming (ICFP), Amsterdam, June 1997*, 1997.

[Myc95] Alan Mycroft, editor. *Proceedings of the Second International Static Analysis Symposium (SAS), Glasgow, Scotland*, number 983 in Lecture Notes in Computer Science. Springer–Verlag, September 25–27 1995.

[Nor98] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In ICFP [ICF98].

[OSW98] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 1998. To appear.

[Par93] Will D. Partain. The `nofib` benchmark suite of Haskell programs. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, page 195, Berlin, 1993. Springer-Verlag.

[PJPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: Moving bindings to give faster programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, New York, 1996. ACM Press.

[PJS98] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[Pot98] François Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, December 1998. Submitted. Available `http://pauillac.inria.fr/~fpottier/publis/publications.html.en`.

[PS96] Jens Palsberg and Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, September 1996.

[San95] André Luís de Mederios Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, July 1995. Available as Technical Report TR-1995-17.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, October 3 1991. Available as DIKU Research Report 92/6.

[TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In R. Cousot and D. A. Schmidt, editors, *Proceedings of the Third International Symposium on Static Analysis (SAS'96), Aachen, Germany*, number 1145 in Lecture Notes in Computer Science, pages 349–265. Springer–Verlag, September 1996.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Conference Record of FPCA'95 SIGPLAN–SIGARCH–WG2.8 Conference on Functional Programming Languages and Computer Architecture, La Jolla, California, 25–28 June 1995*, pages 1–11. ACM, 1995.

[WJ98] Andrew Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Twenty-sixth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'99), January 20–22, 1999, San Antonio, Texas.* ACM Press, 1999.

[Wri96] David A. Wright. Linear, strictness and usage logics. In *Proceedings of CATS'96 (Computing: The Australasian Theory Symposium), Melbourne, Australia, January 29–January 30 1996*, 1996.