Once Upon a Polymorphic Type

Keith Wansbrough[*]
Computing 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 sound type-based 'usage analysis' for a realistic lazy functional language. Accurate information on the usage of program subexpressions in a lazy functional language permits a compiler to perform a number of useful optimisations. However, existing analyses are either *ad-hoc* and approximate, or defined over restricted languages.

Our work extends the *Once Upon A Type* system of Turner, Mossin, and Wadler (FPCA'95). Firstly, we add type polymorphism, an essential feature of typed functional programming languages. Secondly, we include general Haskell-style user-defined algebraic data types. Thirdly, we explain and solve the 'poisoning problem', which causes the earlier analysis to yield poor results. Interesting design choices turn up in each of these areas.

Our analysis is sound with respect to a Launchbury-style operational semantics, and it is straightforward to implement. Good results have been obtained from a prototype implementation, and we are currently integrating the system into the Glasgow Haskell Compiler.

1 Introduction

The knowledge that a value is used at most once is extremely useful to an optimising compiler, because it justifies several beneficial transformations. (We elaborate in Section 2.) Furthermore, it is a property that is invariant across many program transformations, which suggests that the 'used-once' property should be expressed in the value's *type*.

Thus motivated, we present *UsageSP*, a new type system that determines a conservative approximation to the 'used-at-most-once' property (Section 5). Our system builds on existing work, notably *Once Upon A Type* [TWM95a], but makes the following new contributions:

- *We handle a polymorphic language* (Section 6.2).
- *We handle arbitrary, user-defined algebraic data types* (Section 6.4). Built-in data types are handled uniformly with user-defined ones, so there is no penalty for using the latter instead of the former.
- *We identify the 'poisoning problem' and show how to use subsumption to address it*, dramatically increasing the accuracy of usage types (Section 6.3).

The first two extensions are absolutely necessary if the analysis is to be used in practice, since all modern functional languages are polymorphic and have user-defined data types. The extensions required are not routine, and both involve interesting design decisions. All three are also addressed to some degree by the Clean uniqueness-typing system [BS96], in a different but closely-related context (Section 2.2).

Our system comes with:

- A type inference algorithm, so that it can be used as a compiler analysis phase, without ever exposing the type system to the programmer (Section 7).
- A soundness proof that justifies our claimed connection between a value's *type* and its actual *operational uses* (Section 8).

We have a prototype implementation, which confirms that usage-type inference is both relatively simple to implement and computationally cheap. The work we describe here has convinced us that a full-scale implementation would be both straightforward and effective, and so we are currently in the process of adding the analysis to the Glasgow Haskell Compiler.

2 Background and motivation

What is the opportunity that we hope to exploit? In a lazy[1] functional language, bound subexpressions are evaluated only as needed, and never more than once. In order to ensure this, an implementation represents an unevaluated expression by a *thunk*, and overwrites this thunk with its computed value after evaluation for possible reuse. This mechanism involves a great deal of expensive memory traffic: the thunk has to be allocated in the heap, re-loaded into registers when it is evaluated, and overwritten with its value when that is computed.

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

To appear in the *Twenty-sixth ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, January 20–22, 1999, San Antonio, Texas.

[1]In this paper, the term *lazy* refers to the use of a call-by-need semantics [Lau93, PJL92, AFM+95].

Compiler writers therefore seek analyses that help to reduce the cost of thunks. *Strictness analysis* figures out when a thunk is sure to be evaluated *at least once*, thus enabling the replacement of call-by-need with call-by-value [PJ96]. In contrast, this paper describes *usage analysis* which determines when a thunk is sure to be evaluated *at most once*. Such knowledge supports quite a few useful transformations:

Update avoidance. Consider the simple expression $\mathsf{let}\ x = e\ \mathsf{in}\ f\ x$. If it can be demonstrated that f uses the value of x at most once, then x's thunk need not be overwritten after the evaluation of e. The thunk is still constructed, but it is less expensive than before. For implementations that use self-updating thunks, such as the STG machine [PJ92], it is a simple matter to take advantage of such update-avoidance information.

Inlining inside lambdas. Now consider the more complex expression $\mathsf{let}\ x = e\ \mathsf{in}\ \lambda y\ .\ \mathsf{case}\ x\ \mathsf{of}\ \ldots$, and suppose that x does not occur anywhere in the case alternatives. We could avoid the construction of the thunk for x entirely by inlining it at its (single) occurrence site, thus: $\lambda y\ .\ \mathsf{case}\ e\ \mathsf{of}\ \ldots$. Now e is evaluated immediately by the case, instead of first being allocated as a thunk and then later evaluated by the case. Alas, this transformation is, in general, a disaster, because now e is evaluated as often as the lambda is applied, and that might be a great many times. Hence, most compilers pessimistically refrain from inlining redexes inside a lambda. If, however, we could prove that the lambda was applied at most once, and hence that x's thunk would be evaluated at most once, then we could safely perform the transformation.

Floating in. Even when a thunk cannot be eliminated entirely, it may be made less expensive by floating its binding inwards, towards its use site. For example, consider:

$$\mathsf{let}\ x = e\ \mathsf{in}\ \lambda y\ .\ \ldots (f\ (g\ x)) \ldots \tag{1a}$$

If the lambda is known to be called at most once, it would be safe to float the binding for x inwards, thus:

$$\lambda y\ .\ \ldots (f\ (\mathsf{let}\ x = e\ \mathsf{in}\ g\ x)) \ldots \tag{1b}$$

Now the thunk for x may never be constructed (in the case where f does not evaluate its argument); furthermore, if g is strict then we can evaluate e immediately instead of constructing a thunk for it. This transformation is not a *guaranteed* win, because the size of closures can change, but on average it is very worthwhile [PJPS96].

Full laziness. The full laziness transformation hoists invariant sub-expressions out of functions, in the hope of sharing their evaluation between successive applications of the function [PJL91b]. It therefore performs exactly the opposite of the inlining and float-in transformations just discussed above; for example, it would transform (1b) into (1a). As we have just seen, though, hoisting a sub-expression out of a function that is called only once makes things (slightly) worse, not better. Information about the usage of functions can therefore be used to restrict the full laziness transform to cases where it is (more likely to be) of benefit.

The latter two transformations were discussed in detail in [PJPS96], along with measurements demonstrating their effectiveness. That paper pessimistically assumed that every lambda was called more than once. By identifying called-once lambdas, usage information allows more thunks to be floated inwards and fewer to be hoisted outwards, thus improving the effectiveness of both transformations.

To summarise, we have strong reason to believe that accurate information on the usage of subexpressions can allow a compiler to generate significantly better code, primarily by relaxing pessimistic assumptions about lambdas. A great deal more background about transformations that support the compilation of lazy languages is given by [San95], [Gil96], [PJPS96], and [PJ96].

2.1 The problem

So what is the problem? At first one might think that it is quite a simple matter to calculate usage information: simply count syntactic occurrences. But of course this is not enough. Consider $\mathsf{let}\ x = e\ \mathsf{in}\ f\ x$. Even though x occurs syntactically once, whether or not its thunk is evaluated more than once clearly depends on f. The same applies to the free variables of a lambda abstraction:

$$\begin{array}{ll} \mathsf{let} & x = 1 + 2 \\ \mathsf{in} & \mathsf{let}\ \ f = \lambda z\ .\ x + z \\ & \mathsf{in}\ \ \ f\ 3 + f\ 4 \end{array} \tag{2}$$

Here x appears only once in the body of f, but since it is a free variable of f, its value is demanded every time f is applied (here twice). Hence x is here used more than once.

Here is another subtle example, arising from Gill's work on performing `foldr/build` deforestation on the function `foldl` [Gil96, p. 77]. This particular example results from fusion of $foldl\ (+)\ 0\ [1\ ..\ (n-1)]$:

$$\begin{array}{ll} \mathsf{let} & sumUpTo :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ & sumUpTo = \lambda x\ .\ \mathsf{if}\ x < n \\ & \qquad \mathsf{then\ let}\ \ v = sumUpTo\ (x+1) \\ & \qquad\qquad \mathsf{in}\ \ \ \lambda y\ .\ v\ (x+y) \\ & \qquad \mathsf{else}\ \ \lambda y\ .\ y \\ \mathsf{in} & sumUpTo\ 1\ 0 \end{array} \tag{3a}$$

A little inspection should convince you that the inner lambda is called at most once for each application of the outer lambda; this in turn justifies floating out the inner lambda to 'join' the outer lambda, and inlining v, to get the much more efficient code:

$$\begin{array}{ll} \mathsf{let} & sumUpTo :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \\ & sumUpTo = \lambda x\ .\ \lambda y\ .\ \mathsf{if}\ x < n \\ & \qquad \mathsf{then}\ sumUpTo\ (x+1)\ (x+y) \\ & \qquad \mathsf{else}\ \ y \\ \mathsf{in} & sumUpTo\ 1\ 0 \end{array} \tag{3b}$$

What is tricky about this example is that in the original expression it looks as though *sumUpTo* is partially applied to one argument (in the binding for v), and hence perhaps the inner lambda *is* called more than once. Gill used an iterative fixpointing technique to prove that *sumUpTo* is indeed not partially applied.

To summarise, computing usage information in a higher-order language is distinctly non-trivial.

2.2 Previous work

Strictness analysis has a massive literature. Usage analysis, as an optimisation technique for lazy languages, has very little. The first work we know of is that of Goldberg [Gol87], who used abstract interpretation to derive sharing information, which he then used to optimise the generation of supercombinators. This optimisation is essentially equivalent to the improvement we described above under 'full laziness' [PJL91b]. Marlow [Mar93] implemented a similar analysis which has been part of the Glasgow Haskell Compiler since 1993. However the analysis is extremely expensive for some programs, and its results are only used to enable update avoidance, not the other transformations described earlier. A related analysis is presented in [Gil96, pp. 72ff].

A more promising approach is based on types. Compilers go to a great deal of trouble to avoid duplicating work, so a thunk that is evaluated at most once *before* some transformation should also be evaluated at most once *after* that transformation. That in turn suggests that the usage information might be encoded in the thunk's *type*, since a type is conventionally something that is invariant across optimisations.

What is such a 'usage type'? It is tempting to think that a thunk that is used at most once has a *linear type*, in the sense used by the (huge) linear-types literature (*e.g.*, [Gir95, Lin92, Wad93]).

It turns out that most of this work is not immediately applicable, for two reasons. First, linear types are concerned with things that are used *exactly* once, and brook no approximations.[2] Second, linear types are *transitive*, whereas we need an *intransitive* system. For example, consider:

$$\begin{array}{ll} \mathsf{let} & x = e \\ & y = x + 1 \\ \mathsf{in} & y + y \end{array} \tag{4}$$

Here y is certainly used twice, but what about x? A classic linear type system would attribute a non-linear type to x too, but under lazy evaluation x is evaluated only once, on the occasion when y is evaluated for the first time. (When y is evaluated a second time, its already-computed value is used, without reference to x.)

With this in mind, Turner, Mossin, and Wadler in *Once Upon A Type* [TWM95a] presented a type system based on linear logic and on an earlier working paper [LGH$^+$92], but adapted for the usage-type problem we have described. This paper builds directly on their work, but develops it significantly as mentioned in Section 1.

Mogensen [Mog98] presents an extension of the [TWM95a] analysis handling a larger class of data types and adding zero usage annotations. This analysis is significantly more expensive to compute, and does not possess a proof of correctness (indeed, the authors identified several errors in the published version of the system [Mog97]). However, our use of subsumption (Section 6.3) was inspired by this work.

Gustavsson [Gus98] extends Turner *et. al.*'s analysis in a different direction, using a rather lower-level operational semantics that explicitly models update markers on the stack. With this semantics he produces an analysis that aims to reduce the number of update marker checks, as well as avoiding unnecessary updates. Gustavsson's analysis does not treat polymorphism, and its only data structure is the list.

The type rules for our system are similar to those of the uniqueness-type system of Clean [BS96]; both are based on the same linear-logic foundations, augmented with a notion of subytping. The Clean type system identifies 'uniquely-typed' values, whose operational property is that they can be updated in place. This is also a sort of 'used-once' property, but it is, in a sense, dual to ours: a function with a unique argument type places an obligation on the *caller*, but gives an opportunity (for update-in-place) to the *function*. In our system, a function with a used-once argument type places an obligation on the *function*, in exchange for an optimisation opportunity in the *caller*. The systems are dual in the sense that the type rules are similar except that the direction of the subtyping relation is reversed.

Despite the similarity of the type rules, there are numerous differences in detail between our system and that of [BS96]: our system is aimed at a different target, so the proof of soundness is necessarily quite different; our operational semantics is based on a Launchbury-style presentation rather than graph rewriting; we handle full System-F polymorphism; we handle a arbitrarily-nested higher-order language in which lambda abstractions and case expressions can appear anywhere; we lay out the rather subtle design space for the treatment of data structures, rather than making a particular design choice; and there are some technical differences in subtyping rules. Because of these differences, the exact relationship between the two systems is not yet clear. It would be interesting to see whether the duality can be made more precise.

2.3 Plan of attack

A 'usage type' can only encode an *approximation* to the actual usage of a thunk, since the latter is in general undecidable. So the picture we have in mind is this:

- The compiler infers usage types for the whole program. This process can be made computationally tractable by being willing to approximate.
- Guided by this usage information, the compiler now performs many transformations. The usage type information remains truthful across these transformations.
- At any point the compiler may perform usage-type inference again. This may yield better results than before, because some of the transformations may have simplified parts of the program to the point where the (necessarily approximate) inference algorithm can do a better job.
- More transformations can now take place, and so on.

In contrast to systems requiring programmer-supplied annotations, our system is intended as a type-based compiler analysis. The programmer never sees any usage types.

[2]However, *affine* types [Jac94] do permit the kind of approximation made here.

Figure 1 The language of *UsageSP*.

$$\begin{array}{lllll}
\text{Terms} & e & ::= & x & \\
 & & \mid & \lambda x : \sigma \,.\, e & \mid \; e_1\ e_2 \\
 & & \mid & \mathsf{letrec}\ \overline{x_i : \sigma_i = e_i}\ \mathsf{in}\ e & \\
 & & \mid & C\ \overline{\tau_k}\ \overline{e_j} & \\
 & & \mid & \mathsf{case}\ e\ \mathsf{of}\ \overline{C_i \to e_i} & \\
 & & \mid & n & \mid \; e_1 + e_2 \\
 & & \mid & \Lambda \alpha \,.\, e & \mid \; e\ \tau \\
\text{Decls} & T & : & \mathsf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} & \\
\text{Types} & \tau, \phi & ::= & T\ \overline{\tau_k} & \\
\text{(unannotated)} & & \mid & \sigma_1 \to \sigma_2 & \\
 & & \mid & \forall \alpha \,.\, \tau & \\
 & & \mid & \alpha & \\
\text{Types} & \sigma, \rho & ::= & \tau^u & \\
\text{(annotated)} & & & & \\
\text{Usages} & u & ::= & 1 \;\mid\; \omega &
\end{array}$$

The variables x, y, z range over terms; α and β range over types.

3 The language

The language covered by *UsageSP* is presented in Figure 1. Our language is a variant of the Girard–Reynolds polymorphic λ-calculus, extended with many of the features of Core, the intermediate language of the Glasgow Haskell Compiler (GHC) [GT98].

Compared with the language of [TWM95a], *UsageSP* replaces nil and cons with general constructors and a corresponding general case construct, adds type abstraction and application, and requires explicit types on lambda abstractions and let bindings. In addition, it permits the application of functions to expressions (rather than only to variables). We write $\overline{e_i}$ to abbreviate $e_1, e_2, \ldots, e_n$.

The form of the case expression may require some explanation. The conventional expression $\mathsf{case}\ e\ \mathsf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}$ both selects an alternative and binds variables; thus there are usually three constructs that bind variables: lambda, letrec, and case. The form of case we use here avoids binding variables, and instead passes the constructor arguments to the selected expression as function arguments to be bound by a lambda. This slightly complicates the statement of the typing rule for case, but simplifies our proofs.

Figure 1 also gives the syntax of types. Notice their two-level structure. A σ-type bears a *usage annotation* (1 or ω), indicating the use made of a value of that type; a τ-type does not. The annotations denote 'used at most once' and 'possibly used many times', respectively. We define a function $|\cdot|$ to obtain the (outermost) annotation on a σ-type: $|\tau^u| = u$.

Int and other primitive data types are treated as nullary, predeclared type constructors. For arrow types and Int the system is equivalent to that of [TWM95a]. For lists and other algebraic data types, however, there is an annotation on the type itself but not on the type arguments (*e.g.*, $(\mathsf{List\ Int})^\omega$); this choice differs from [TWM95a] and is discussed further in Section 6.4.

Figure 2 The restricted language.

$$\begin{array}{lllll}
\text{Terms} & e & ::= & a & \\
\text{(normalised)} & & \mid & \lambda x : \sigma \,.\, e & \mid \; e\ a \\
 & & \mid & \mathsf{letrec}\ \overline{x_i : \sigma_i = e_i}\ \mathsf{in}\ e & \\
 & & \mid & C\ \overline{\tau_k}\ \overline{a_j} & \\
 & & \mid & \mathsf{case}\ e\ \mathsf{of}\ \overline{C_i \to e_i} & \\
 & & \mid & n & \mid \; e_1 + e_2 \\
 & & \mid & \Lambda \alpha \,.\, e & \mid \; e\ \tau \\
\text{Atoms} & a & ::= & x & \\
 & & \mid & a\ \tau & \\
\text{Values} & v & ::= & \lambda x : \sigma \,.\, e & \mid \; C\ \overline{\tau_k}\ \overline{a_j} \\
 & & \mid & n & \mid \; \Lambda \alpha \,.\, v
\end{array}$$

Two key design decisions relate to the handling of polymorphism. Firstly, a type abstraction abstracts an *unannotated* (τ-) type, rather than a σ-type: $\forall \alpha \,.\, \tau$ rather than $\forall \alpha \,.\, \sigma$. Secondly, type variables α range over τ-types, rather than over σ-types. That is, the argument of a type application is a τ-type, and the arguments of a type constructor are also τ-types. The reason for these decisions will become clear shortly: the first decision is a consequence of the operational semantics and is discussed in Section 4.2; the second is deeper and is discussed in Section 6.2.

4 Operational semantics

Our analysis is operationally motivated, so it is essential to formalise the operational semantics of our language. We base this on that of [TWM95a] (which is itself based on the standard operational semantics for lazy functional languages, [Lau93]), with alterations and extensions to handle user-defined data types and polymorphism, and to support our proofs. The natural (big-step) semantics is presented in Figure 3, and is largely conventional.

H denotes a *heap* consisting of typed bindings of the form $(x : \sigma) \mapsto e$. The heap is unordered; all variables are distinct and the bindings may be mutually recursive. A *configuration* $\langle H \rangle\, e$ denotes the expression e in the context provided by the heap H; variables free in e are bound in H. The big-step reduction $\langle H_1 \rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2 \rangle\, v$ signifies that given the heap H_1 the expression e reduces to the value v, leaving the heap H_2 (values are defined in Figure 2). The significance of the type context $\overline{\alpha_k}$ is explained in Section 4.2. Standard capture-free substitution of the variable y for the variable x in expression e is denoted $e[x := y]$; similarly substitution of the type τ for the type variable α everywhere in expression e is denoted $e[\alpha := \tau]$. Type substitution extends pointwise to heaps.

The reduction relation $\Downarrow_{\cdot}$ is defined over a language slightly smaller than that of Figure 1; this is given in Figure 2. Following Launchbury [Lau93] we use letrec bindings to name arguments of applications and constructors, in order to preserve sharing. The translation into the smaller language is straightforward. We allow applications of functions and constructors to atoms rather than just variables so the substitutions in ($\Downarrow$-LetRec) can work (see Section 4.2).

Figure 3 Natural semantics for *UsageSP*.

$$\frac{|\sigma| = 1 \qquad \langle H_1\rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, v}{\langle H_1, x : \sigma \mapsto e\rangle\, x \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, v}\ (\Downarrow\text{-}\textsc{Var-Once}) \qquad \frac{|\sigma| = \omega \qquad \langle H_1\rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, v}{\langle H_1, x : \sigma \mapsto e\rangle\, x \Downarrow_{\overline{\alpha_k}} \langle H_2, x : \sigma \mapsto v\rangle\, v}\ (\Downarrow\text{-}\textsc{Var-Many})$$

$$\frac{\begin{array}{c}\text{fresh } \overline{y_i} \qquad S = (\overline{x_j := y_j\ \overline{\alpha_k}}) \\ \langle H_1, \overline{y_i : (\forall \overline{\alpha_k}\,.\,\tau_i)^{u_i} \mapsto \Lambda \overline{\alpha_k}\,.\, e_i[S]}\rangle\, e[S] \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, v\end{array}}{\langle H_1\rangle\, \mathsf{letrec}\ \overline{x_i : \tau_i{}^{u_i} = e_i}\ \mathsf{in}\ e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, v}\ (\Downarrow\text{-}\textsc{LetRec})$$

$$\frac{}{\langle H\rangle\, \lambda x : \sigma\,.\, e \Downarrow_{\overline{\alpha_k}} \langle H\rangle\, \lambda x : \sigma\,.\, e}\ (\Downarrow\text{-}\textsc{Abs}) \qquad \frac{\langle H_1\rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, \lambda x : \sigma\,.\, e' \qquad \langle H_2\rangle\, e'[x := a] \Downarrow_{\overline{\alpha_k}} \langle H_3\rangle\, v}{\langle H_1\rangle\, e\ a \Downarrow_{\overline{\alpha_k}} \langle H_3\rangle\, v}\ (\Downarrow\text{-}\textsc{App})$$

$$\frac{}{\langle H\rangle\, C\ \overline{\tau_k}\ \overline{a_j} \Downarrow_{\overline{\alpha_k}} \langle H\rangle\, C\ \overline{\tau_k}\ \overline{a_j}}\ (\Downarrow\text{-}\textsc{Con}) \qquad \frac{\langle H_1\rangle\, e_1 \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, n_1 \qquad \langle H_2\rangle\, e_2 \Downarrow_{\overline{\alpha_k}} \langle H_3\rangle\, n_2}{\langle H_1\rangle\, e_1 + e_2 \Downarrow_{\overline{\alpha_k}} \langle H_3\rangle\, n_1 \oplus n_2}\ (\Downarrow\text{-}\textsc{PrimOp})$$

$$\frac{\langle H_1\rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, C_j\ \overline{\tau_k}\ a_1 \ldots a_m \qquad \langle H_2\rangle\, e_j\ a_1 \ldots a_m \Downarrow_{\overline{\alpha_k}} \langle H_3\rangle\, v}{\langle H_1\rangle\, \mathsf{case}\ e\ \mathsf{of}\ \overline{C_i \to e_i} \Downarrow_{\overline{\alpha_k}} \langle H_3\rangle\, v}\ (\Downarrow\text{-}\textsc{Case}) \qquad \frac{}{\langle H\rangle\, n \Downarrow_{\overline{\alpha_k}} \langle H\rangle\, n}\ (\Downarrow\text{-}\textsc{Int})$$

$$\frac{\text{fresh } \alpha' \qquad \langle H_1\rangle\, e[\alpha := \alpha'] \Downarrow_{\overline{\alpha_k}, \alpha'} \langle H_2\rangle\, v}{\langle H_1\rangle\, \Lambda\alpha\,.\, e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, \Lambda\alpha'\,.\, v}\ (\Downarrow\text{-}\textsc{TyAbs}) \qquad \frac{\langle H_1\rangle\, e \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, \Lambda\alpha\,.\, v}{\langle H_1\rangle\, e\ \tau \Downarrow_{\overline{\alpha_k}} \langle H_2\rangle\, v[\alpha := \tau]}\ (\Downarrow\text{-}\textsc{TyApp})$$

4.1 Usage control

A conventional semantics would have only one rule for variables, the ($\Downarrow$-Var-Many) rule. This rule performs the usual call-by-need overwriting of the heap binding. But as in [TWM95a] we restrict this rule to apply only when the variable's usage annotation is ω; when the annotation is 1 a second rule, ($\Downarrow$-Var-Once), applies. This latter rule deletes the binding after use; hence an incorrect usage annotation will lead to a *stuck expression* to which no reduction rule applies. The proof in Section 8 shows that the annotations we generate can never result in a stuck expression, and hence guarantees that our usage annotations are correct with respect to the operational semantics.

4.2 Type information

As in many implementations of typed functional languages, GHC erases all type information prior to program execution; thus type abstractions and applications do not normally appear in the operational semantics.

In order to support our proofs, however, our operational semantics *does* carry type information in expressions and inside the heap. This information is entirely ignored by the semantics, apart from the topmost annotation on bound variables which is used as described above to guide the application of the ($\Downarrow$-Var-Once) and ($\Downarrow$-Var-Many) rules. Indeed, deleting this type information and merging the two rules yields a semantics essentially the same as Launchbury's original semantics [Lau93].

This leads to the unusual form of the type rules ($\Downarrow$-TyAbs) and ($\Downarrow$-TyApp). These rules define type abstractions and applications as being 'transparent': evaluation *is* permitted underneath a type abstraction (hence only a type abstraction of a *value* is a value according to the definition in Figure 2), and a type application performs type substitution *after* the abstraction has been evaluated. There is an exactly equivalent operational semantics involving no types (other than the topmost annotations on bound variables), and in this 'erased' operational semantics ($\Downarrow$-TyAbs) and ($\Downarrow$-TyApp) do precisely nothing. The presence of types in the operational semantics is purely to aid the proofs in Section 8.

While evaluating underneath a type lambda, we keep track of the type context by an annotation $\overline{\alpha_k}$ on the reduction relation. This type context is simply an ordered list of the variables bound by enclosing type lambdas. We abbreviate $\Downarrow_{\{\}}$ by $\Downarrow$.

The type context is manipulated only in ($\Downarrow$-TyAbs) and ($\Downarrow$-LetRec). The former rule simply adjusts the annotation while evaluating under a type lambda; it applies both to $e\ \tau$ and to $a\ \tau$ since an atom a is also an expression. The latter rule maintains the invariant that heap bindings never have free type variables, using a technique similar to that used in GHC for let-floating in the presence of type lambdas. Before placing a binding in the heap, potentially-free type variables are bound by fresh type lambdas; a substitution is performed to ensure that the types remain correct.[3]

Notice that this yields call-by-*name* behaviour for type applications. We do not share partial applications $e\ \tau$. This is as we expect, since the partial applications have no real operational significance. The alternative would require creating an extra thunk for $e\ \tau$ in order to share it, distinct from

[3]The notation $\forall \overline{\alpha_k}\,.\,e$ abbreviates $\forall \alpha_1\,.\,\forall \alpha_2\,.\,\ldots e$; similarly for type lambdas and applications.

the thunk for e. This does not correspond to the behaviour of the evaluator.

The 'transparent' nature of the ($\Downarrow$-TyAbs) and ($\Downarrow$-TyApp) rules determines the design decision mentioned in Section 3 regarding the syntax of polymorphic types. All expressions must be given a σ-type. Say expression e has type $\sigma_1 = \tau^u$, and the expression $\Lambda\alpha\ .\ e$ has type σ_2. Since type abstraction has no operational significance and usage is an operational property, we expect that $|\sigma_1| = |\sigma_2|$. Were σ_2 to be $(\forall\alpha\ .\ \tau^u)^{u'}$ (*i.e.*, were type abstraction to abstract over a σ-type), we would have two usage annotations u and u' constrained always to have the same value. Instead, we define them to be the same variable by defining type abstraction to be over a τ-type: $\sigma_2 = (\forall\alpha\ .\ \tau)^u$.

Notice that in a language featuring intensional polymorphism [HM95] this decision would be different. Typing information would have operational significance, types would be retained at run time, and the ($\Downarrow$-TyAbs) and ($\Downarrow$-TyApp) rules would appear differently; hence type abstraction would be over a σ-type, reflecting the distinct usages of the type abstraction and its body (u and u' in the example above).

5 Type system

In this section and the next we present a type system that approximates which subexpressions are used more than once during evaluation, according to the operational semantics. The typing judgements of the system are shown in Figure 5.

5.1 Usages and usage constraints

Usage annotations are as defined in Figure 1, and are permitted to be 1 (used at most once) or ω (possibly used many times). Usages are ordered, with $1 < \omega$. Constraints on usage annotations are either a relational constraint $u \leq u'$ using this ordering, or a simple equality constraint $u = \omega$.

5.2 Subtypes

A subtype ordering $\preccurlyeq$ on both σ- and τ-types is obtained as the obvious extension of $\leq$. Since we wish to consider $\preccurlyeq$ as a subtyping relation, however, the ordering under this relation is opposite to the ordering of annotations. For example, $1 < \omega$ and hence $\mathsf{Int}^\omega \preccurlyeq \mathsf{Int}^1$. This is because a value that is permitted to be used more than once (*e.g.*, Int^ω) may safely be used in a context that consumes its argument at most once (*e.g.*, Int^1). Subtyping is used in Section 6.3 to solve the so-called 'poisoning problem'.

The subtyping relation is defined inductively in Figure 4. The ordering is contravariant on function types. Universally-quantified types are related by unifying quantified variables (we treat types as α-equivalence classes, so this is implicit). Saturated type constructors[4] are related if and only if all instantiated constructor argument types are related. The relation $\alpha \in fv^\varepsilon(\tau)$ states that type variable α has a free

[4]Our language fragment handles only saturated type constructors, thus remaining essentially in System F rather than F_ω. There is a straightforward but pessimistic approximation for higher-order type constructors: if the type constructor is a variable α rather than a constant T, simply assume that all its arguments occur both covariantly and contravariantly.

Figure 4 The subtyping relation $\cdot \preccurlyeq \cdot$.

$$\frac{u_2 \leq u_1 \qquad \tau_1 \preccurlyeq \tau_2}{{\tau_1}^{u_1} \preccurlyeq {\tau_2}^{u_2}}\ (\preccurlyeq\text{-Annot})$$

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

$$\frac{}{\alpha \preccurlyeq \alpha}\ (\preccurlyeq\text{-TyVar}) \qquad \frac{\tau_1 \preccurlyeq \tau_2}{\forall\alpha\ .\ \tau_1 \preccurlyeq \forall\alpha\ .\ \tau_2}\ (\preccurlyeq\text{-ForAll})$$

$$\frac{\begin{array}{c}\mathsf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} \\ \alpha_k \in fv^\varepsilon(\tau_{ij}) \Rightarrow \tau_k \preccurlyeq^\varepsilon \tau'_k \quad \text{for all } k,\ \varepsilon,\ i,\ j\end{array}}{T\ \overline{\tau_k} \preccurlyeq T\ \overline{\tau'_k}}\ (\preccurlyeq\text{-TyCon})$$

$$\text{where } \tau_k \preccurlyeq^\varepsilon \tau'_k \text{ is } \begin{cases}\tau_k \preccurlyeq \tau'_k & \text{if } \varepsilon = + \\ \tau'_k \preccurlyeq \tau_k & \text{if } \varepsilon = -\end{cases}$$

ε-ve occurrence in τ. The subtyping relation is discussed further in Section 7.1.

5.3 Occurrences

To aid in the statement of the type rules, we define a *syntactic occurrence* function. The expression $occur(x, e)$ gives the number of free syntactic occurrences of the variable x in the expression e. It is defined inductively. The definition is trivial except for the case statement, for which we define

$$occur(x, \mathsf{case}\ e\ \mathsf{of}\ \overline{C_i \to e_i}) = occur(x, e) + \mathbf{max}_{i=1}^{n}\, occur(x, e_i)$$

Here we conservatively approximate by taking the maximum number of syntactic occurrences in any alternative.

Turner *et. al.* avoid using this function in their technical report [TWM95b], and instead detect multiple occurrences through special operators for combining contexts. This necessitates tighter control over context manipulation; our approach is exactly equivalent, but leads to a simpler inference algorithm and easier proofs.

5.4 Contexts

A context Γ is a set of term and type variables, the former annotated with their types. Standard rules define well-formed contexts and well-kinded types; these are omitted here (refer for example to [SP94]). All term and type variables appearing in the context are distinct.

We write $\Gamma, x : \sigma$ for the extension of the set Γ with the element $(x : \sigma)$, and Γ, α for the extension of Γ with the element α. $x \in \Gamma$ expresses that the term variable x is found in the context Γ (with some unspecified type), and $\alpha \in \Gamma$ that the type variable α is found in the context Γ. Since the term variables in a context are all distinct, Γ can be seen as a function from term variables to their types: $\Gamma(x)$ is the type of term variable x.

Figure 5 Type rules for *UsageSP*.

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (\vdash\text{-Var}) \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}^u}\ (\vdash\text{-Int})$$

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

$$\frac{\begin{array}{c}\Gamma, \overline{x_j : \sigma_j} \vdash e_i : \sigma_i' \qquad \sigma_i' \preccurlyeq \sigma_i \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j} \vdash e : \sigma \\ \mathit{occur}(x_i, e) + \sum_{j=1}^{n} \mathit{occur}(x_i, e_j) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i\end{array}}{\Gamma \vdash \mathsf{letrec}\ \overline{x_i : \sigma_i = e_i}\ \mathsf{in}\ e : \sigma}\ (\vdash\text{-LetRec}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int}^{u_1} \qquad \Gamma \vdash e_2 : \mathsf{Int}^{u_2}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}^{u_3}}\ (\vdash\text{-PrimOp})$$

$$\frac{\begin{array}{c}\mathsf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} \\ \Gamma \vdash e_j : \sigma_{ij}' \quad \sigma_{ij}' \preccurlyeq (\tau_{ij}[\overline{\alpha_k := \tau_k}])^u \quad \text{for all } j\end{array}}{\Gamma \vdash C_i\ \overline{\tau_k}\ \overline{e_j} : (T\ \overline{\tau_k})^u}\ (\vdash\text{-Con}) \qquad \frac{\begin{array}{c}\mathsf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} \\ \Gamma \vdash e : (T\ \overline{\tau_k})^u \\ \Gamma \vdash e_i : \sigma_i' \quad \sigma_i' \preccurlyeq (\overline{(\tau_{ij}[\overline{\alpha_k := \tau_k}])^u} \to \sigma)^1 \quad \text{for all } i\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \overline{C_i \to e_i} : \sigma}\ (\vdash\text{-Case})$$

$$\frac{\Gamma, \alpha \vdash e : \tau^u}{\Gamma \vdash \Lambda\alpha \,.\, e : (\forall \alpha \,.\, \tau)^u}\ (\vdash\text{-TyAbs}) \qquad \frac{\Gamma \vdash e : (\forall \alpha \,.\, \tau_2)^u}{\Gamma \vdash e\ \tau_1 : (\tau_2[\alpha := \tau_1])^u}\ (\vdash\text{-TyApp})$$

5.5 Type rules

The type rules for *UsageSP* are given in Figure 5. The judgement form $\Gamma \vdash e : \sigma$ states that in context Γ the expression e has type σ. Usage constraints appearing above the line in the rules are interpreted as meta-constraints on valid derivations. Notice that an expression may have multiple incomparable typings: for example, the term $\lambda x : \mathsf{Int} \,.\, x$ may be typed as either $\mathsf{Int}^1 \to^\omega \mathsf{Int}^1$ or $\mathsf{Int}^\omega \to^\omega \mathsf{Int}^\omega$. For this reason, the system of Figure 5 does not enjoy principal types. However our use of subsumption means we can still assign each variable a type that 'fits' every use of that variable; indeed we show in Section 7 that we can choose an 'optimal' such type. We also describe a system that employs usage polymorphism to recover principal types, in Section 6.3.

The notation is conventional, and the rules are essentially the same as the usual rules for a lambda calculus with subsumption (see, *e.g.*, [CG94]) except that we omit bounded quantification. Notice that the rules have been presented here in syntax-directed form: each rule corresponds to an alternative in the term syntax presented in Figure 1.

The most complex rule is ($\vdash$-Case). In the subtype constraint, the substitution $\tau_{ij}[\overline{\alpha_k := \tau_k}]$ instantiates the appropriate component type from the data type declaration, and the notation $(\overline{\sigma_i} \to \sigma)^1$ is shorthand for the used-once multiple-argument application $(\sigma_1 \to (\sigma_2 \to (\cdots \to (\sigma_n \to \sigma)^1 \cdots)^1)^1)^1$. Notice that because of subsumption each alternative may accept a supertype of the actual type of each component, and may return a subtype of the result of the entire case expression.

6 UsageSP

Having presented the 'boilerplate' of our type system, we now introduce its less-standard features.

6.1 Multiple uses

Contexts are treated in an entirely standard way—the rules of weakening and contraction are implicit in the presentation. How then do we identify multiple uses?

The answer lies in the appropriate use of the syntactic occurrence function defined in Section 5.3. If a variable occurs more than once in its scope, the variable is certainly used more than once and we annotate it thus in ($\vdash$-Abs) and ($\vdash$-LetRec). Additionally, recall Example 2 from Section 2.1: each time an abstraction is used, any free variables of that abstraction may also be used. Thus all free variables of an abstraction have at least the usage of the abstraction itself. This appears as the third line of the premise in ($\vdash$-Abs).

The ($\vdash$-LetRec) rule includes a condition on syntactic occurrences that recognises both recursive and nonrecursive uses of variables, and handles them appropriately.

Finally, a use of a constructor entails a (possible) use of its contents. Thus we require in ($\vdash$-Con) that an expression placed into a constructor has a usage at least that of the constructor application itself (expressed by placing u from the constructor type as annotation on the τ-type of the constructor argument from the data type declaration). This corresponds to [TWM95a]'s "global well-formedness condition". If one thinks of the standard functional encoding of a constructor, then this constraint is similar to that discussed above on the free variables of a lambda.

6.2 Type polymorphism

The first development we mentioned in Section 1 was the extension to a polymorphic language. Handling polymorphism in some way is clearly essential for realistic applications. This is the rôle of the ($\vdash$-TyAbs) and ($\vdash$-TyApp) rules (Figure 5). It is not the case that introducing *type polymorphism* requires introducing *usage polymorphism* as well; the two are in fact orthogonal issues. We consider type polymorphism in this section and usage polymorphism in the next.

There are two design decisions to be considered in the implementation of type polymorphism. The first concerns the ($\vdash$-TyAbs) rule, and the representation of a type abstraction. Given that $\Gamma, \alpha \vdash e : \sigma$, what type has $\Lambda\alpha \,.\, e$ in context Γ? Clearly for consistency it must be a σ type; we expect something of the form $(\forall\alpha \,.\, \cdot)^{\cdot}$.

In Section 4.2 we argue that for operational reasons it only makes sense to have a single usage annotation on a type abstraction, and thus $\Gamma \vdash \Lambda\alpha \,.\, e : (\forall\alpha \,.\, \tau)^u$ for some τ and u. The same argument shows $u = |\sigma|$, and thus it follows logically that $\tau^u = \sigma$. This gives us the ($\vdash$-TyAbs) rule shown; it can be interpreted as lifting the usage annotation of the original expression outside the abstraction.

The second design decision concerns the ($\vdash$-TyApp) rule, and the range of a type variable. Type application should reverse the behaviour of ($\vdash$-TyAbs): given $\Gamma \vdash e : (\forall\alpha \,.\, \tau)^u$, we expect to have $\Gamma \vdash e\ \psi : (\tau[\alpha := \psi])^u$. But what should ψ (and hence α) be: a τ- or a σ-type?

The answer becomes obvious when we consider that the purpose of the usage typing of a function is to convey information about how that function uses its arguments. Consider the functions $dup : \forall\alpha \,.\, \alpha \to Pair\ \alpha\ \alpha$, which duplicates its argument, making a pair with two identical components, and $inl : \forall\alpha \,.\, \alpha \to L\ \alpha$, which injects its argument into a sum type. We should be able to express that *dup* uses its argument more than once, whereas *inl* does not; that is, we should have $dup : \forall\alpha \,.\, \alpha^\omega \to \ldots$ but $inl : \forall\alpha \,.\, \alpha^1 \to \ldots$. It is only possible so to annotate α if $\alpha \in \tau$.

The alternative is to introduce bounded quantification, so that we have $dup : \forall\alpha : |\alpha| \geq \omega \,.\, \alpha \to \ldots$ and $inl : \forall\alpha : |\alpha| \geq 1 \,.\, \alpha \to \ldots$, as suggested in [TWM95a, §4.5]. However this leads to a loss of precision: consider the function

$$twoone = \Lambda\alpha \,.\, \lambda x : \alpha \,.\, \lambda y : \alpha \,.\, (x, x, y) \tag{5}$$

We want here to express that although both arguments have the same underlying *type*, their *usages* differ: the first is used twice, the second once: $twoone : \forall\alpha \,.\, \alpha^\omega \to \alpha^1 \to \ldots$. Bounded quantification is insufficient for this; we must instead allow α to range over unannotated types and add *separate* annotations on each occurrence. This is done in the ($\vdash$-TyApp) rule shown.

6.3 Subsumption and usage polymorphism

A distinctive feature of our system compared to more conventional linear type systems, including the [TWM95a] analysis, is our use of *subsumption*. In this section we discuss and justify this choice. Consider the expression

$$\begin{array}{ll} \mathsf{let} & f = \lambda x \,.\, x + 1 \\ & a = 2 + 3 \\ & b = 5 + 6 \\ \mathsf{in} & a + (f\ a) + (f\ b) \end{array} \tag{6}$$

Here it is clear by inspection that a's value will be demanded twice, but b's only once. The implementations we have in mind use *self-updating thunks* [PJ92], so that f's behaviour is independent of whether its argument thunk requires to be updated. We therefore expect b to be given a used-once type. (In an implementation where the *consumer* of a thunk performs the update, f could not be applied to both a used-once and a used-many thunk. Exploiting sharing information is much harder in such implementations.)

However, [TWM95a] infers the type Int^ω, denoting possible multiple usage, for b. Why? Clearly a's type must be Int^ω, since a is used more than once. So a non-subsumptive type system must attribute the type $\mathsf{Int}^\omega \to \ldots$ to f. But since f is applied to b as well, b gets f's argument type Int^ω. We call this the 'poisoning problem', because one call to f 'poisons' all the others. Poisoning is absolutely unacceptable in practice — for a start, separate compilation means that we may not know what all the calls to f are.

There are two solutions to the poisoning problem. One possibility is to use *usage polymorphism.* That is, f gets the type $\forall u \,.\, \mathsf{Int}^u \to \ldots$, so that it is applicable to both values of type Int^ω (by instantiating u to ω), and to values of type Int^1 (by instantiating u to 1). Such a system was sketched by [TWM95a], although as a solution to a different problem. We see three difficulties with this approach:

- *Technically,* simple polymorphism turns out to be insufficient; bounded polymorphism is required, with types such as $(\forall u_1 \,.\, \forall u_2 \leq u_1 \,.\, \forall u_3 \leq u_1 \,.\, \forall\alpha \,.\, (List\ \alpha)^{u_1} \to ((List\ \alpha)^{u_2} \to (List\ \alpha)^{u_2})^{u_3})^\omega$.
- *Theoretically,* bounded usage polymorphism significantly complicates the (already hard) problem of providing a denotational-semantic model for the intermediate language.
- *Pragmatically,* we fear that the compiler will get bogged down in the usage lambda abstractions, applications, and constraints, that must accompany usage polymorphism in a type-based compiler. Even quite modest functions or data constructors can be polymorphic in literally dozens of usage variables, so they quickly become quite dominant in a program text.

Furthermore, bounded polymorphism seems like a sledgehammer to crack a nut. Suppose f has the type $\mathsf{Int}^1 \to \ldots$. We want to interpret this as saying that f evaluates its argument at most once, but saying nothing about how f is used. In particular, it should be perfectly OK to apply f to an argument either of type Int^1 or of type Int^ω without fuss. On the other hand, if g's type is $\mathsf{Int}^\omega \to \ldots$, we interpret that as saying that g may evaluate its argument more than once, and so it should be ill-typed to apply g to an argument of type Int^1.

This one-way notion of compatibility is just what subtyping was invented for. Using a subtyping judgement based on usages (see Section 5.2), the addition of a subsumption rule to the system:

$$\frac{\Gamma \vdash e : \sigma' \qquad \sigma' \preccurlyeq \sigma}{\Gamma \vdash e : \sigma}\ (\vdash\text{-Sub})$$

at one stroke removes the poisoning problem. In terms of the example above, ($\vdash$-Sub) allows us to conclude that an argument of type Int^ω also has type Int^1 and hence can be an argument of f. We have pushed subsumption down into ($\vdash$-App), ($\vdash$-Con), ($\vdash$-LetRec), and ($\vdash$-Case) in Figure 5 in order to obtain a syntax-directed rule set. This brings the rules closer to the form of a type inference algorithm (Section 7) and makes the proofs easier.

Returning to Example 6 at the start of this section, we see that a still gets annotation ω, but this does not now affect f's argument annotation of 1 and thus does not affect the annotation given to b, which is 1 as we originally expected. The inference of an ω usage of a variable will no longer result in the needless poisoning of the remainder of the program.

All of this is pretty simple. There is no clutter in the term language corresponding to subsumption, so the compiler is not burdened with extra administration. The system is simple enough that all our proofs include subsumption, whereas none of the proofs in [TWM95a] cover the system with usage polymorphism. Lastly, the effectiveness of subsumption seems to be closely tied up with our use of self-updating thunks; we wonder whether there may be some deeper principle hiding inside this observation.

Does the polymorphic sledgehammer crack any nuts that subsumption does not? Certainly it does. For a start, it enables us to recover a conventional principal typing property. Furthermore, usage polymorphism is able to express the dependency of usage information among several arguments. For example, consider the *apply* function:

$$\mathit{apply} = \lambda f\, x\, .\, f\ x \tag{7}$$

The most general type our system can infer for *apply*, assuming no partial applications of *apply* are permitted, is

$$\mathit{apply} :: \forall\alpha\, .\, \forall\beta\, .\, ((\alpha^{\omega} \to \beta^{\omega})^{1} \to (\alpha^{\omega} \to \beta^{\omega})^{1})^{\omega}$$

In the application (*apply negate x*), our system would therefore infer that x might be evaluated more than once when actually it is evaluated exactly once. (We are assuming that *negate* evaluates its argument just once.) Usage polymorphism is able to express the connection between the two arguments to *apply* by giving it the type

$$\mathit{apply} :: \forall u_1\, .\, \forall u_2\, .\, \forall\alpha\, .\, \forall\beta\, .\, ((\alpha^{u_1} \to \beta^{u_2})^{1} \to (\alpha^{u_1} \to \beta^{u_2})^{1})^{\omega}$$

The situation is not unlike that for strictness analysis, where abstract values are often *widened* to make the analysis more tractable, but at the expense of losing information about the interaction of function arguments. Our present view is that the costs (in terms of complexity and compiler efficiency) of usage polymorphism are likely to be great, while the benefits (in terms of more accurate types) are likely to be small. It remains an interesting topic for further work.

6.4 Data structures

The extension of the system to handle general Haskell-style data types with declarations of the form $\mathsf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}}$ is not entirely straightforward.

For a start, should the constructor component types specified in the data declaration range over τ- or σ-types? At first it seems they should range over σ-types, as do the arguments of functions. But a little thought reveals that it does not make sense to talk about how a constructor uses its arguments: their usage depends on the way in which the constructed data is used, not on the constructor itself. Hence, we give no annotation to constructor argument types in the data declaration; instead we infer the usage of constructor arguments at a constructor aplication from the way in which the constructed value is used. Constructors do not, therefore, have a single functional type; instead there is a special typing rule for construction.

The key issue, therefore, is *how to ensure multiple use of a component (via a* case *expression) propagates to the appropriate constructor site*, forcing the annotation of the appropriate constructor argument to be ω. The manner of this propagation gives rise to a number of alternatives.

Consider the code fragment[5]

$$\begin{array}{ll} \mathsf{data}\ T\ \alpha = \mathit{MkT}\ \alpha\ (\alpha^{\omega} \to \alpha^{\omega}) & \\ \mathsf{let}\ \ a :: \mathsf{Int}^{?} = 1 + 1 & (8) \\ \qquad f :: (\mathsf{Int}^{\omega} \to \mathsf{Int}^{\omega})^{?} = \lambda y\, .\, y + y & \\ \qquad x :: (T\ \mathsf{Int})^{?} = \mathit{MkT}\ a\ f & \\ \qquad g :: (\forall\alpha\, .\, (T\ \alpha)^{?} \to \alpha^{\omega})^{1} & \\ \qquad g\ (\mathit{MkT}\ a\ f) = f\ (f\ a) & \\ \mathsf{in}\ \ \ g\ x & \end{array}$$

Here the function g uses its argument ($\mathit{MkT}\ a\ f$) once, but the components of the argument are used respectively once and twice. Clearly this information must propagate to the type of x and thence to the types of a and f. We consider four alternative ways of typing this expression to propagate this information:

- *We could explicitly specify constructor component usage annotations in the type.* This would give g the type $(\forall\alpha\, .\, (T\ 1\ \omega\ \alpha)^{1} \to \alpha^{\omega})^{1}$, where we introduce usage arguments to T giving the uses of each component. Now x is given the type $(T\ 1\ \omega\ \mathsf{Int})^{1}$, and a and f annotations 1 and ω respectively.

  We reject this for the same reasons as before (Section 6.3); it amounts to usage polymorphism and would significantly complicate our system technically, theoretically, and pragmatically.

- An intermediate alternative, more *ad hoc* but perhaps more practical, would be to follow the example of the Clean type system [BS96] and attach usage annotations to each type argument of the constructor. These annotations would then be applied uniformly to all occurrences of that type variable, and some other rule would be used to provide the remaining annotations. Thus g would have the type $(\forall\alpha\, .\, (T\ \alpha^{1})^{1} \to \alpha^{\omega})^{1}$, x the type $(T\ \mathsf{Int}^{1})^{1}$, and a and f annotations 1 and ω respectively. The rule would here have to assign annotation ω to the function argument.

  We have already argued in Section 6.2 that usage annotations should not be linked to specific type variables. Furthermore, the use of a separate rule to assign non-type-variable annotations adds extra complexity.

- *We could assume that all constructor components are used more than once.* This seems reasonable since one usually places data in a data structure in order that it may be used repeatedly. This would give g the type

[5]Notice that the syntax still forces us to give annotations for the 'internal' types of the constructor arguments. The only reasonable choice seems to be the most-general annotation, ω, since we do not want to restrict the arguments to which the constructor can be applied, and in the presence of separate compilation we cannot see all its uses.

$(\forall \alpha \,.\, (T\ \alpha)^1 \to \alpha^\omega)^1$, x the type $(T\ \mathsf{Int})^1$, and both a and f the annotation ω.

While this avoids the overhead of usage polymorphism, we lose a lot of precision by making this assumption. Consider the common use of pairs to return two values from a function (say quotient and remainder from a division). The pair and its components are each used only once; if the rules above are used they will be annotated ω and thunks will be built for them needlessly.

- *We could identify the constructor's overall usage annotation with the annotations on its components.* Since one component (f) of g's argument is used more than once, this means we annotate the argument itself with ω; so g is given the type $(\forall \alpha \,.\, (T\ \alpha)^\omega \to \alpha^\omega)^1$, x the type $(T\ \mathsf{Int})^\omega$, and both a and f the annotation ω.

  Since we use only a single usage annotation, if any component is used more than once, all are annotated ω. This identification preserves good behaviour for, *e.g.*, pairs used once, while behaving only slightly worse in the general case. Since different construction sites may have different annotations, the effect of the approximation is localised.

The final alternative above seems a reasonable compromise between simplicity and precision: we avoid adding usage polymorphism and its attendant complexity, at the expense of sub-optimal types in some cases. The general type rules ($\vdash$-CON) and ($\vdash$-CASE) in Figure 5 follow this alternative. The rules are uniform and do not penalise users for defining their own data types: built-in data types such as lists and pairs are handled by the same rules as user-defined ones.

7 Inference

In order to use *UsageSP* as a compiler analysis phase, it is necessary to provide an inference algorithm. Such an algorithm must be decidable, and should infer annotations that are in some sense optimal. It should also be fast.

In general, type inference in System F is undecidable [Wel94]. But we are already in possession of a fully-typed term in a language lacking only usage annotations; we need only infer the optimal values of these annotations. This task is much easier. The annotation sites are uniquely determined by the structure of the unannotated term, and there is a unique type derivation for our annotated term according the the rules of Figure 5 (this is why we chose to present the rules in syntax-directed form). Thus inference proceeds in two steps:

1. Given an unannotated term e and an initial context Γ, we examine its type derivation and collect all the usage constraints in a constraint set, Θ. This is straightforward apart from the computation of the subtyping relation $\preccurlyeq$, which we discuss in Section 7.1.
2. We then obtain the optimal solution of Θ (in a sense defined below), and apply it to e to yield an optimally-annotated term. We discuss this in Section 7.2.

This two-pass technique, separating the generation and solution of the constraints, is possible because all the usage variables are global; inference of bounded usage quantifiers (Section 6.3) would probably require local constraint solution to occur during type inference.

7.1 Subtyping

A naïve definition of the subtyping relation would state that

$$\frac{\mathsf{data}\ T\ \overline{\alpha_k} = \overline{C_i\ \overline{\tau_{ij}}} \qquad \tau_{ij}[\overline{\alpha_k := \tau_k}] \preccurlyeq \tau_{ij}[\overline{\alpha_k := \phi_k}] \quad \text{for all } i,\, j}{T\ \overline{\tau_k} \preccurlyeq T\ \overline{\phi_k}} \quad \text{(TYCON-NAÏVE)}$$

However, this definition is not well-founded. A recursive data type (*e.g.*, $List\,\alpha$) contains an instance of the head type ($List\,\alpha$) as one of the constructor argument types τ_{ij}, and so the same clause appears in the premise and the conclusion of the ($\preccurlyeq$-TYCON-NAÏVE) rule. It follows that a straightforward use of this rule to generate usage constraints would go into an infinite loop. This behaviour is fairly simple to avoid in the case of regular data types [AC93], but Haskell admits *non-regular* (also known as *nested* [BM98, Oka98]) data types in which a type constructor may appear recursively with different arguments from the head occurrence.

In order to compute the subtyping relation for general Haskell data types, we use the rule originally presented, ($\preccurlyeq$-TYCON). It turns out that (TYCON-NAÏVE) holds, as a reverse implication, in this system; this fact is used in the proof of the substitution lemma (Lemma B.2).

The relation $\alpha \in fv^\varepsilon(\sigma)$ can be defined straightforwardly by induction on the structure of σ. In practice $\bigvee_{ij} (\alpha_k \in fv^\varepsilon(\tau_{ij}))$ can be precomputed for all T, k, ε very early in the compiler, since it depends only on the data type definitions and not on the code. Membership in the subtyping relation can then be computed in the obvious recursive fashion.

7.2 Constraint solution

The constraint set Θ characterises all usage variable substitutions satisfying the type rules; in fact there is a non-standard notion of principal type based on Θ, formalised in [TWM95a]. The constraint set comprises constraints of two forms, either $u \leq u'$ or $u = \omega$, and defines a partial order on the set of usage variables extended with maximal and minimal elements ω and 1 respectively; a solution must respect this partial order.

We want the *optimal* solution, in the sense that we want as many 1-annotations as possible. This clearly exists: we let every usage variable not equal to ω be set equal to 1. Finding the solution is straightforward due to the trivial nature of the constraints, and can be done in time linear in the number of constraints.

Due to the syntax-directed nature of the rules and the explicit typing, the inference generates a constraint set of size approximately linear[6] in the size of the program text (assuming data type declarations of constant size). Hence the analysis overall is approximately linear in the size of the program text.

[6]The approximation arises from the ($\vdash$-CASE) rule; the subtyping of the result type generates constraints proportional to the result type (which is not specified in the program text). Hence arbitrary nesting of case expressions inside case alternatives can yield arbitrarily large constraint sets. We do not expect this situation to arise in practice, but even in this case the constraint set is at most quadratic in the size of the program text.

Figure 6 Extension of type rules to include configurations.

$$\frac{\begin{array}{c} H = \overline{x_i : \sigma_i \mapsto e_i} \\ \Gamma, \overline{x_j : \sigma_j} \vdash e_i : \sigma'_i \qquad \sigma'_i \preccurlyeq \sigma_i \quad \text{for all } i \\ \Gamma, \overline{x_j : \sigma_j}, \overline{\alpha_k} \vdash e : \sigma \\ \mathit{occur}(x_i, e) + \sum_{j=1}^{n} \mathit{occur}(x_i, e_j) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{array}}{\Gamma \,;\, \overline{\alpha_k} \vdash_{\mathit{Conf}} \langle H \rangle\, e : \sigma} \quad (\vdash\text{-}\textsc{Conf})$$

8 Syntactic soundness

It is important to ensure that our type rules do indeed correspond to the operational behaviour of the evaluator. Since our operational semantics deletes 1-annotated variables from the heap after use (Section 4.1), an incorrect annotation of the program will eventually result in a stuck expression: a variable will be referenced that is not present in the heap, and reduction will not be able to proceed. Hence, *if we can demonstrate that a well-typed term never gets stuck, then we will know that our annotations are truthful.*

Our plan for the proof is as follows. We use the technique of Wright and Felleisen [WF94]:

1. We introduce a new typing judgement, $\vdash_{\mathit{Conf}}$, that specifies when a configuration (Section 4) is well-typed (Figure 6). A configuration is essentially a letrec, and the type rule is essentially identical to ($\vdash$-LetRec). However, notice the additional type context $\overline{\alpha_k}$; these variables (representing type lambdas under which we are currently evaluating) scope over the expression but *not* over the heap, since the heap is not permitted to contain unbound type variables (see Section 4.2).

2. We demonstrate *subject reduction*: if a well-typed term is reducible, it reduces to a term that is itself well-typed, as a subtype of the original type. In order to do this we first derive from our natural semantics $\Downarrow$ (Figure 3) a set of small-step reduction rules $\rightarrow$. We present these quite conventional rules in Appendix A and prove soundness and adequacy with respect to the natural semantics. Given these rules, subject reduction takes the form of the following theorem:

   Theorem 8.1 (Subject reduction)
   If $;\vdash_{\mathit{Conf}} \langle H \rangle\, e : \sigma$ *and* $\langle H \rangle\, e \rightarrow \langle H' \rangle\, e'$, *then* $;\vdash_{\mathit{Conf}} \langle H' \rangle\, e' : \sigma'$, *where* $\sigma' \preccurlyeq \sigma$.

3. Finally, we demonstrate *progress*: a well-typed term is either reducible or a value; it cannot be stuck.

   Theorem 8.2 (Progress)
   For all configurations $\langle H \rangle\, e$ *such that* $;\vdash_{\mathit{Conf}} \langle H \rangle\, e : \sigma$, *either there exist* H' *and* e' *such that* $\langle H \rangle\, e \rightarrow \langle H' \rangle\, e'$, *or* e *is a value.*

Since our system possesses subject reduction and progress, it is *type-safe*: the types are respected by the operational semantics, and our annotations are truthful.

The proofs follow the proofs of [TWM95b], with a number of modifications and additions (notably progress). We give proof sketches in Appendix B; full proofs may be found in the companion technical report [WPJ98].

9 Status and further work

We have presented a new usage type system, *UsageSP*, which handles a language essentially equivalent to the intermediate language of the Glasgow Haskell Compiler. We have proved this type system sound with respect to the operational semantics of the language. In addition, we have presented an efficient and optimal type inference algorithm for this system, enabling it to be hidden from the programmer. We have yet to prove this algorithm correct, but we expect the proof to be straightforward.

Our prototype implementation is implemented in approximately 2000 non-comment lines of Haskell code, 800 of which is the inference proper. For comparison, the GHC compiler [GT98] is approximately 50 000 lines of Haskell; the strictness analyser is about 1400 lines.

It was useful to develop the prototype in parallel with the type system itself; several of the design decisions made in the evolution of the type system were clarified and motivated by our practical experience with the prototype.

There are a number of areas for future work.

- We expect it to be reasonably straightforward to integrate the new analysis into the Glasgow Haskell Compiler. However, integration with GHC will force us to address a number of issues. GHC permits separate compilation. We propose to use a 'pessimising' translation on the inferred types of exported functions, giving them the most generally-applicable types possible.

  GHC also has unboxed types [PJL91a]. It is not yet clear whether our analysis should treat these differently from other data types.

  It is possible to infer better types for multi-argument functions if we are sure they are never partially applied. Our implementation of *UsageSP* could possibly be extended to use a version of the worker/wrapper transformation currently used by the strictness analyser [PJL91a], yielding improved usage annotations in common cases.

- Mogensen [Mog98] augments his analysis with a zero-usage annotation. Intuitively, zero-usages should occur infrequently; programmers are unlikely to write such expressions and a syntactic dead-code analysis in the compiler removes any that arise during optimisation. However Mogensen shows that such information is useful when it refers to portions of a data structure.

  The full ramifications of adding a zero annotation to our analysis are currently unclear. Certain portions of it would require rethinking. For example, a *syntactic* occurrence function is no longer sufficient: a variable may occur syntactically in a subexpression which is used zero times, and thus *not* occur for the purposes of the analysis. Additionally, as noted in footnote 7, some significant modifications would have to be made to our proof technique. Constraint solution over the larger class of constraints generated by such a system is also more complex, possibly asymptotically. It is unclear whether the advantages of greater precision would outweigh the disadvantages of a significantly more complicated and expensive analysis.

Acknowledgements

We wish especially to thank David N. Turner for a series of discussions in which key ideas of this paper arose. We also wish to thank Clem Baker–Finch, Erik Barendsen, Gavin Bierman, Neil Ghani, Simon Marlow, Tom Melham, Torben Mogensen, Chris Okasaki, Valeria de Paiva, Benjamin Pierce, Andrew Pitts, Eike Ritter, Mark Shields, Sjaak Smetsers, Philip Wadler, Limsoon Wong, and the anonymous referees for many other useful discussions and suggestions.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[ACM95] *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), San Francisco, California*, 1995. ACM Press.

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In ACM [ACM95].

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Proc. 4th Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, number 1422 in LNCS, pages 52–67. Springer–Verlag, June 1998.

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

[CG94] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object Oriented Programming*, Foundations of Computing, chapter 8, pages 247–292. MIT Press, 1994.

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

[Gir95] Jean-Yves Girard. Linear logic: Its syntax and semantics. In Girard, Lafont, and Regnier, eds, *Advances in Linear Logic*, no. 222 in London Math. Soc. Lect. Notes Series. Cambridge University Press, 1995.

[Gol87] Benjamin Goldberg. Detecting sharing of partial applications in functional programs. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, number 274 in Lecture Notes in Computer Science, pages 408–425. Springer–Verlag, September 1987.

[GT98] The GHC Team. The Glasgow Haskell Compiler user's guide, version 3.02. Distributed with GHC. Available `http://www.dcs.gla.ac.uk/fp/software/ghc/`, April 1998.

[Gus98] Jörgen Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, 1998.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In ACM [ACM95].

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

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina*, pages 144–154. ACM Press, 1993.

[LGH+92] John Launchbury, Andy Gill, John Hughes, Simon Marlow, Simon Peyton Jones, and Philip Wadler. Avoiding unnecessary updates. In *Proc. 5th Glasgow Functional Programming Workshop*, 1992.

[Lin92] Patrick Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

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

[Mog97] Torben Æ. Mogensen. Types for 0, 1 or many uses. In Chris Clack, Tony Davie, and Kevin Hammond, eds, *Proc. 9th International Workshop on Implementation of Functional Languages, St. Andrews, Scotland, September 10th–12th, 1997*, pages 157–165, 1997.

[Mog98] Torben Æ. Mogensen. Types for 0, 1 or many uses. Updated and corrected version of [Mog97]. Obtained from author, 1998.

[Oka98] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PJ96] Simon L. Peyton Jones. Compilation by transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP '96: 6th European Symposium on Programming, Linköping, Sweden, April 1996*, number 1058 in Lecture Notes in Computer Science, pages 18–44. Springer-Verlag, 1996.

[PJL91a] Simon L. Peyton Jones and John Launchbury. Unboxed values as first-class citizens in a non-strict functional language. In *Proceedings of the 1991 Confernce on Functional Programming Languages and Computer Architecture*, September 1991.

[PJL91b] Simon L. Peyton Jones and David Lester. A modular fully-lazy lambda lifter in Haskell. *Software – Practice and Experience*, 21(5):479–506, May 1991.

[PJL92] Simon L. Peyton Jones and David Lester. *Implementing Functional Languages: A Tutorial.* Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[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.

[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.

[SP94] Martin Steffen and Benjamin Pierce. Higher-order subtyping. Technical Report LFCS-ECS-94-280, University of Edinburgh, February 1994. Also Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94. Revised version with Fun subtyping.

[TWM95a] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Conf. on Functional Programming Languages and Computer Architecture (FPCA'95), La Jolla, California*, pages 1–11, 1995.

Figure 8 Evaluation contexts.

$$\begin{array}{llll} \text{Contexts} & E & ::= & [\,]\,a \\ & & \mid & \mathsf{case}\ [\,]\ \mathsf{of}\ \overline{C_i \to e_i} \\ & & \mid & [\,] + e \quad \mid \quad n + [\,] \\ & & \mid & [\,]\,\tau \end{array}$$

[TWM95b] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. Technical Report TR-1995-8, Computing Science Department, University of Glasgow, 1995. Extended version of [TWM95a]. Available `http://www.dcs.gla.ac.uk/people/old-users/dnt/MossinTurnerWadlerTR95.dvi.gz`.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science, Gdansk, August–September 1993, proceedings*, number 711 in Lecture Notes in Computer Science. Springer–Verlag, 1993.

[Wel94] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proc. 9th Ann. IEEE Symp. on Logic in Computer Science (LICS)*, pp. 176–185, Paris, 1994.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WPJ98] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. Technical Report TR-1998-19, Department of Computing Science, University of Glasgow, 1998.

A Reduction rules

As explained in Section 8, to support our subject reduction proof we derive from our natural semantics $\Downarrow$ (Figure 3) a set of small-step reduction rules $\to$. These appear in Figure 7, and are unsurprising. The ($\to$-CTX) rule collapses a number of rules that define when we may evaluate a subterm, using evaluation contexts E as defined in Figure 8. These reduction rules are provably equivalent to the natural semantics given earlier. Full proofs appear in the companion technical report [WPJ98].

B Syntactic soundness

Subject reduction and progress are discussed in Section 8. In this appendix we sketch the proofs of these results; full proofs appear in the companion technical report [WPJ98].[7]

We commence by proving several lemmas dealing with special cases arising in the proof of the main theorem. Note that in the following we use the convention that $H = (\overline{x_i : \sigma_i \mapsto e_i})$ and $H' = (\overline{y_i : \rho_i \mapsto e'_i})$. Also recall that $\to$ abbreviates $\to_{\{\}}$.

Lemma B.1 (Context pruning)
If $\Gamma, y : \sigma_0 \vdash e : \sigma$ and $occur(y, e) = 0$, then $\Gamma \vdash e : \sigma$.

Context pruning allows us to remove an unused variable from the context. It is used to handle the ($\to$-VAR-ONCE) rule.

Lemma B.2 (Substitution)
If $\Gamma, y : \sigma_0, y' : \sigma'_0 \vdash e : \sigma$ where $\sigma'_0 \preccurlyeq \sigma_0$ and $occur(y', e) > 0 \Rightarrow |\sigma'_0| = \omega$, then $\Gamma, y' : \sigma'_0 \vdash e[y := y'] : \sigma'$ where $\sigma' \preccurlyeq \sigma$.

Substitution is a key result, showing that if we substitute one variable for another, the new a subtype of the old, the expression is still well-typed and has a type that is a subtype of the original type. This lemma is used for the ($\to$-APP) rule.

Lemma B.3 (Nondeletion)
If $occur(x_i, e) + \sum_{j=1}^{n} occur(x_i, e_j) > 0 \Rightarrow |\sigma_i| = \omega$ for some i and $\langle H \rangle\, e \to_{\overline{\alpha_k}} \langle H' \rangle\, e'$ then there is some $y_k \equiv x_i$ in H', and $occur(y_k, e') + \sum_{j=1}^{n} occur(y_k, e'_j) > 0 \Rightarrow |\rho_k| = \omega$ as before.

The nondeletion lemma guarantees that essential bindings are never dropped while evaluating inside an evaluation context (rule ($\to$-CTX)).

Given the above lemmas, we show subject reduction. We do this by means of an invariance lemma which strengthens the inductive hypothesis. This is to handle the case ($\to$-VAR-EVAL) where we must evaluate a binding inside the heap: the binding is temporarily removed from the heap, but since the expression may recursively refer to itself the variable must be placed in the context to retain well-typedness.

Lemma B.4 (Invariance)
If $\Gamma ; \overline{\alpha_k} \vdash_{Conf} \langle H \rangle\, e : \sigma$ and $\langle H \rangle\, e \to_{\overline{\alpha_k}} \langle H' \rangle\, e'$, then $\Gamma ; \overline{\alpha_k} \vdash_{Conf} \langle H' \rangle\, e' : \sigma'$, where $\sigma' \preccurlyeq \sigma$.

Proof Proof is by induction on the depth of the inference of $\langle H \rangle\, e \to_{\overline{\alpha_k}} \langle H' \rangle\, e'$, making key use of Lemmas B.1, B.2 and B.3. We present here two key cases from the proof: ($\to$-VAR-ONCE) and ($\to$-TYABS).

case ($\to$-VAR-ONCE): $\langle H, x : \sigma \mapsto e \rangle\, x \to_{\overline{\alpha_k}} \langle H \rangle\, e$ where $|\sigma| = 1$

By assumption we have that $\Gamma ; \overline{\alpha_k} \vdash_{Conf} \langle H, x : \sigma \mapsto e \rangle\, x : \sigma$ (it is easy to show by ($\vdash$-CONF) and ($\vdash$-VAR) that it is the same σ; we will omit this demonstration here and in the subsequent two cases). Hence by ($\vdash$-CONF) we have that

$$\Gamma, \overline{x_j : \sigma_j}, x : \sigma \vdash e_i : \sigma'_i \qquad \sigma'_i \preccurlyeq \sigma_i \quad \text{for all } i \qquad (9)$$

$$\Gamma, \overline{x_j : \sigma_j}, x : \sigma, \overline{\alpha_k} \vdash e : \sigma' \qquad \sigma' \preccurlyeq \sigma \qquad (10)$$

[7] Notice that our proof technique would be inadequate for a system that tracked zero-usages. In such a system, we could infer that x is used only once in the expression $\langle x : \mathsf{Int}^1 \mapsto e_1, y : \mathsf{Int}^1 \mapsto e_2 \rangle\, \mathit{fst}\ (x, y) + \mathit{snd}\ (x, y)$; this reduces in two steps to an expression $\langle y : \mathsf{Int}^1 \mapsto e_2 \rangle\, e_1 + \mathit{snd}\ (x, y)$ in which x appears without a binding in the heap, thus breaking our current definition of well-typedness. However, this could be remedied by the addition of a type rule

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma \qquad |\sigma_1| = 0}{\Gamma \vdash e : \sigma}\ (\vdash\text{-ZERO})$$

The full ramifications of introducing zero-usages, however, are currently unclear.

Figure 7 Reduction rules for *UsageSP*.

$$\frac{|\sigma| = 1}{\langle H, x : \sigma \mapsto e\rangle\, x \rightarrow_{\overline{\alpha_k}} \langle H\rangle\, e}\ (\rightarrow\text{-}\textsc{Var-Once}) \qquad \frac{|\sigma| = \omega}{\langle H, x : \sigma \mapsto v\rangle\, x \rightarrow_{\overline{\alpha_k}} \langle H, x : \sigma \mapsto v\rangle\, v}\ (\rightarrow\text{-}\textsc{Var-Many})$$

$$\frac{|\sigma| = \omega \qquad \langle H\rangle\, e \rightarrow_{\overline{\alpha_k}} \langle H'\rangle\, e'}{\langle H, x : \sigma \mapsto e\rangle\, x \rightarrow_{\overline{\alpha_k}} \langle H', x : \sigma \mapsto e'\rangle\, x}\ (\rightarrow\text{-}\textsc{Var-Eval}) \qquad \frac{\langle H\rangle\, e \rightarrow_{\overline{\alpha_k}} \langle H'\rangle\, e'}{\langle H\rangle\, E[e] \rightarrow_{\overline{\alpha_k}} \langle H'\rangle\, E[e']}\ (\rightarrow\text{-}\textsc{Ctx})$$

$$\frac{\text{fresh } \overline{y_i} \qquad S = (\overline{x_j := y_j\ \overline{\alpha_k}})}{\langle H\rangle\ \mathsf{letrec}\ \overline{x_i : {\tau_i}^{u_i} = e_i}\ \mathsf{in}\ e \rightarrow_{\overline{\alpha_k}} \langle H, \overline{y_i : (\forall \overline{\alpha_k}\ .\ \tau_i)^{u_i} \mapsto \Lambda \overline{\alpha_k}\ .\ e_i[S]}\rangle\, e[S]}\ (\rightarrow\text{-}\textsc{LetRec})$$

$$\frac{}{\langle H\rangle\, (\lambda x : \sigma\ .\ e)\ a \rightarrow_{\overline{\alpha_k}} \langle H\rangle\, e[x := a]}\ (\rightarrow\text{-}\textsc{App}) \qquad \frac{}{\langle H\rangle\, n_1 + n_2 \rightarrow_{\overline{\alpha_k}} \langle H\rangle\, n_1 \oplus n_2}\ (\rightarrow\text{-}\textsc{PrimOp})$$

$$\frac{}{\langle H\rangle\ \mathsf{case}\ C_j\ \overline{\tau_k}\ a_1 \ldots a_m\ \mathsf{of}\ \overline{C_i \rightarrow e_i} \rightarrow_{\overline{\alpha_k}} \langle H\rangle\, e_j\ a_1 \ldots a_m}\ (\rightarrow\text{-}\textsc{Case})$$

$$\frac{\text{fresh } \alpha' \qquad \langle H\rangle\, e[\alpha := \alpha'] \rightarrow_{\overline{\alpha_k}, \alpha'} \langle H'\rangle\, e'}{\langle H\rangle\, \Lambda\alpha\ .\ e \rightarrow_{\overline{\alpha_k}} \langle H'\rangle\, \Lambda\alpha'\ .\ e'}\ (\rightarrow\text{-}\textsc{TyAbs}) \qquad \frac{}{\langle H\rangle\, (\Lambda\alpha\ .\ v)\ \tau \rightarrow_{\overline{\alpha_k}} \langle H\rangle\, v[\alpha := \tau]}\ (\rightarrow\text{-}\textsc{TyApp})$$

$$\begin{aligned} &\mathit{occur}(x_i, x) + \textstyle\sum_{j=1}^{n} \mathit{occur}(x_i, e_j) + \mathit{occur}(x_i, e) > 1 \\ &\qquad \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{aligned} \tag{11}$$

$$\begin{aligned} &\mathit{occur}(x, x) + \textstyle\sum_{j=1}^{n} \mathit{occur}(x, e_j) + \mathit{occur}(x, e) > 1 \\ &\qquad \Rightarrow |\sigma| = \omega \end{aligned} \tag{12}$$

By (12), the assumption that $|\sigma| = 1$, and the fact that $\mathit{occur}(x, x) = 1$, we have that for all i $\mathit{occur}(x, e_i) = \mathit{occur}(x, e) = 0$. Hence, by Lemma B.1, (9) and (10) reduce to

$$\Gamma, \overline{x_j : \sigma_j} \vdash e_i : \sigma_i' \qquad \sigma_i' \preccurlyeq \sigma_i \quad \text{for all } i \tag{13}$$

$$\Gamma, \overline{x_j : \sigma_j}, \overline{\alpha_k} \vdash e : \sigma' \qquad \sigma' \preccurlyeq \sigma \tag{14}$$

Since $\mathit{occur}(x_i, x) = 0$ for all i, we have from (11) that

$$\textstyle\sum_{j=1}^{n} \mathit{occur}(x_i, e_j) + \mathit{occur}(x_i, e) > 1 \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \tag{15}$$

and combining (13), (14), and (15) we have by ($\vdash$-Conf) the result we require, $\Gamma\,;\, \overline{\alpha_k} \vdash_{\mathit{Conf}} \langle H\rangle\, e : \sigma'$ where $\sigma' \preccurlyeq \sigma$.

case ($\rightarrow$-TyAbs): $\langle H\rangle\, \Lambda\alpha\ .\ e \rightarrow_{\overline{\alpha_k}} \langle H'\rangle\, \Lambda\alpha'\ .\ e'$ where $\langle H\rangle\, e[\alpha := \alpha'] \rightarrow_{\overline{\alpha_k}, \alpha'} \langle H'\rangle\, e'$, α' fresh

By assumption and ($\vdash$-TyAbs) we have that $\Gamma\,;\, \overline{\alpha_k} \vdash_{\mathit{Conf}} \langle H\rangle\, \Lambda\alpha\ .\ e : (\forall\alpha\ .\ \tau)^u$ Hence by ($\vdash$-Conf) we have that

$$\Gamma, \overline{x_j : \sigma_j} \vdash e_i : \sigma_i' \qquad \sigma_i' \preccurlyeq \sigma_i \quad \text{for all } i \tag{16}$$

$$\Gamma, \overline{x_j : \sigma_j}, \overline{\alpha_k} \vdash \Lambda\alpha\ .\ e : (\forall\alpha\ .\ \tau)^u \tag{17}$$

$$\begin{aligned} &\mathit{occur}(x_i, \Lambda\alpha\ .\ e) + \textstyle\sum_{j=1}^{n} \mathit{occur}(x_i, e_j) > 1 \\ &\qquad \Rightarrow |\sigma_i| = \omega \quad \text{for all } i \end{aligned} \tag{18}$$

By ($\vdash$-TyAbs) from (17) we have that

$$\Gamma, \overline{x_j : \sigma_j}, \overline{\alpha_k}, \alpha' \vdash e[\alpha := \alpha'] : \tau^u[\alpha := \alpha'] \tag{19}$$

Since $\mathit{occur}(x_i, e[\alpha := \alpha']) = \mathit{occur}(x_i, \Lambda\alpha'\ .\ e)$, (16), (19), and (18) imply by ($\vdash$-Conf) that $\Gamma\,;\, \overline{\alpha_k}, \alpha' \vdash_{\mathit{Conf}} \langle H\rangle\, e[\alpha := \alpha'] : \tau^u[\alpha := \alpha']$. By the inductive hypothesis this implies that

$$\Gamma\,;\, \overline{\alpha_k}, \alpha' \vdash_{\mathit{Conf}} \langle H'\rangle\, e' : {\tau'}^{u'} \quad \text{where } {\tau'}^{u'} \preccurlyeq \tau^u[\alpha := \alpha'] \tag{20}$$

Hence by ($\vdash$-Conf) we have that

$$\Gamma, \overline{y_j : \rho_j} \vdash e_i' : \rho_i' \qquad \rho_i' \preccurlyeq \rho_i \quad \text{for all } i \tag{21}$$

$$\Gamma, \overline{y_j : \rho_j}, \overline{\alpha_k}, \alpha' \vdash e' : {\tau'}^{u'} \tag{22}$$

$$\begin{aligned} &\mathit{occur}(y_i, e') + \textstyle\sum_{j=1}^{n} \mathit{occur}(y_i, e_j') > 1 \\ &\qquad \Rightarrow |\rho_i| = \omega \quad \text{for all } i \end{aligned} \tag{23}$$

By ($\vdash$-TyAbs) from (22) we have that

$$\Gamma, \overline{y_j : \rho_j}, \overline{\alpha_k} \vdash \Lambda\alpha'\ .\ e' : (\forall\alpha'\ .\ \tau')^{u'} \tag{24}$$

and clearly $(\forall\alpha'\ .\ \tau')^{u'} \preccurlyeq (\forall\alpha'\ .\ \tau[\alpha := \alpha'])^u \equiv_\alpha (\forall\alpha\ .\ \tau)^u$.

Since $\mathit{occur}(y_i, e') = \mathit{occur}(y_i, \forall\alpha'\ .\ e')$, (21), (24), and (23) imply by ($\vdash$-Conf) that $\Gamma\,;\, \overline{\alpha_k} \vdash_{\mathit{Conf}} \langle H\rangle\, \Lambda\alpha'\ .\ e : (\forall\alpha'\ .\ \tau')^{u'}$, where $(\forall\alpha'\ .\ \tau')^{u'} \preccurlyeq (\forall\alpha\ .\ \tau)^u$ as required.

The remaining cases are similar. □

Subject reduction (Theorem 8.1) follows immediately from invariance: let $\Gamma = \emptyset$. Progress (Theorem 8.2) is clear by inspection of the reduction and type rules.