Once upon a type*

In *7'th International Conference on Functional Programming and Computer Architecture*, San Diego, California, June 1995.

David N. Turner Philip Wadler
University of Glasgow†
dnt,wadler@dcs.glasgow.ac.uk

Christian Mossin
University of Copenhagen‡
mossin@diku.dk.

Abstract

A number of useful optimisations are enabled if we can determine when a value is accessed at most once. We extend the Hindley-Milner type system with *uses*, yielding a type-inference based program analysis which determines when values are accessed at most once. Our analysis can handle higher-order functions and data structures, and admits principal types for terms.

Unlike previous analyses, we prove our analysis sound with respect to call-by-need reduction. Call-by-name reduction does *not* provide an accurate model of how often a value is used during lazy evaluation, since it duplicates work which would actually be shared in a real implementation.

Our type system can easily be modified to analyse usage in a call-by-value language.

1 Introduction

This paper describes a method for determining when a value is used at most once. Our method is based on a simple modification of the Hindley-Milner type system. Each type is labelled to indicate whether the corresponding value is used at most once, or may possibly be used many times.

Our type system has a number of applications:

Program transformation: If it is determined that a variable is accessed at most once, then one may safely inline the expression bound to the variable without reducing efficiency. In particular, we can determine when it is safe to inline an expression into the body of a function.

Avoiding closure update: Implementations of lazy languages use updates to share the evaluation of closures. If it is determined that a closure is accessed at most once, then there is no need to overwrite the closure with the result of evaluation.

Enabling data structure update: If it is determined that a structure such as a 'cons' cell or an array is accessed at most once, then the structure may safely be updated in place.

The last of these application areas has received considerable attention, the second of these some attention, and the first almost none. This is quite surprising, since expression inlining is central to a wide range of program transformations. In particular, our method provides a sound basis for a number of transformations that were previously dealt with in an ad hoc manner in the Glasgow Haskell Compiler.

Previous analyses to determine when a value is used at most once have been based on either call-by-name reduction (Wright and Baker-Finch [WB93], Courtenage and Clack [CC94]) or call-by-need reduction (Launchbury *et al.* [Lau92], Marlow [Mar93]). The call-by-name analyses have been proved sound, but are not well-suited for optimisation of lazy languages. Our analysis is the first call-by-need analysis to be proved sound, and sometimes provides more accurate information than other call-by-need analyses. Our proof of soundness is based on the operational semantics of Launchbury [Lau93] and the call-by-need calculus of Ariola *et al.* [AFMOW95].

The type system presented here is based on ideas taken from the linear logic of Girard [Gir87] and its successor the Logic of Unity [Gir93]. However, it turns out to be convenient to present this work without reference to linear logic. Some of the connections are traced in a companion paper [MOTW95], which relates linear logic to the call-by-need calculus of Ariola *et al.*

We modify the Hindley-Milner type system [Hin69, DM82] by attaching *uses* to types. Type judgements include a constraint set relating uses, similar to the constraint sets relating subtypes in the work of Mitchell [Mit84, Mit91]. As with the Hindley-Milner system, there is an algorithm that determines a principal type for an expression. Annotating types with usage information also provides a convenient mechanism for communicating usage information across module boundaries, since typed languages such as Haskell already import type information from separately-compiled modules.

A small modification to our analysis enables it to determine when variables are used *exactly* once (as opposed to at most once), making it suitable for use with call-by-value (as opposed to call-by-need) evaluation.

In a companion technical report [MTW95] we present an extension of this analysis which can handle full recursion (we also give details of proofs and include a description of our type inference algorithm).

*This work was supported by EPSRC (GR/J 527099), 'Save space with linear types', ESPRIT (6809) 'Semantique', ESPRIT (ECUS025) 'Atlantique', and the Danish Technical Research Council.

†Computing Science Department, University of Glasgow, Glasgow G12 8QQ, Scotland.

‡DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark. This work was performed while Mossin was visiting the University of Glasgow.

1.1 The problem

Some examples may help to illustrate the nature of the problem solved.

We wish to attach *uses* to values. The use 1 indicates that a value is used at most once, while the use ω indicates that a value may be used any number of times.

Consider the following. (Example 1.)

$$\begin{array}{l} \text{let } x = 1 + 2 \text{ in} \\ \quad \text{let } y = x + 3 \text{ in} \\ \quad\quad y + y \end{array}$$

Here it is safe to replace x by $1 + 2$ within the body of the outer 'let'. But it is not safe to replace y by $x + 3$ within the body of the inner 'let', as the resulting program would compute $x + 3$ twice rather than once. Our type system attaches use 1 to x and use ω to y.

Our argument depends crucially on the use of a call-by-need semantics. Under a call-by-name semantics $x+3$ is always computed twice. A call-by-name analysis must therefore attach a use equivalent to ω to both x and y, showing why such analyses are not suited for our purpose.

At first glance, it may seem child's play to determine if a value is used at most once under call-by-need. Surely, if a variable appears at most once in a program, then the value it is bound to is used at most once? In fact, this is not the case.

Consider the following. (Example 2.)

$$\begin{array}{l} \text{let } x = 1 + 2 \text{ in} \\ \quad \text{let } f = \lambda z.\, x + z \text{ in} \\ \quad\quad f\,3 + f\,4 \end{array}$$

Even though x appears only once in the body of the outer 'let', replacing x by $1+2$ is unsafe, as the resulting program will compute $1+2$ twice rather than once. Our type system attaches use ω to both x and f.

Consider the following. (Example 3.)

$$\begin{array}{l} \text{let } x = 1 + 2 \text{ in} \\ \quad \text{let } f = (\text{let } y = x + 3 \text{ in } \lambda z.\, y + z) \text{ in} \\ \quad\quad f\,4 + f\,5 \end{array}$$

Here, again, one can safely replace the one occurrence of x by $1 + 2$, although it may require a moment's thought to convince oneself this is the case. Indeed, this example is sufficiently difficult that the analyses proposed by Launchbury et al. [Lau92] and by Marlow [Mar93] are both overly conservative, and in effect attach use ω to x. However, our type system attaches use 1 to x and use ω to y and f.

1.2 Call-by-need

Our work is based on the operational semantics of call-by-need proposed by Launchbury [Lau93], and on the call-by-need lambda calculus of Ariola *et al.* [AFMOW95]. A correspondence between these two approaches has already been shown in the latter work.

Launchbury's rules include an explicit treatment of closure update. By modifying his rules to allow some closures to be non-updating, we verify that our type system can be used to avoid unnecessary closure updates.

The soundness of our type system is established by showing that it satisfies a subject reduction property: applying a call-by-need reduction to a term leaves its type unchanged, including type information regarding usage.

Launchbury restricts functions to be applied to variables, while Ariola *et al.* allow functions to be applied to arbitrary expressions. As we explain in Section 5, the difference between these approaches is significant for our chosen implementation, the Glasgow Haskell Compiler [PHHPW93], which is closely based on the STG-machine of Peyton Jones [Pey92]. Therefore, in this paper we adopt Launchbury's syntax (which was influenced by the Haskell compiler and the STG-machine), and adapt the results of Ariola *et al.* to it.

1.3 Program transformation

If we want to use program transformation as the basis of efficient compilation of a functional language, it is not only important that transformation preserves meaning but also that the transformed program executes at least as fast as the original.

Consider Church's beta rule:

$$(\lambda x.\, e_1)\, e_2 \Longrightarrow [e_2/x]e_1.$$

This rule is good in that it eliminates one application step, but bad in that it may duplicate some computation. (In particular, computation of e_2 may be duplicated if x is used more than once in e_1.)

The call-by-need calculus of Ariola *et al.* addresses this problem by modifying the above rule:

$$(\lambda x.\, e_1)\, e_2 \Longrightarrow \text{let } x = e_2 \text{ in } e_1.$$

This rule, together with a number of rules for manipulating 'let', allow us to safely transform programs, without the risk of duplicating work.

However, there are a number of transformations that are useful and safe and which are not part of the call-by-need calculus. The most important of these is that the beta rule

$$(\lambda x.\, e_1)\, e_2 \Longrightarrow [e_2/x]e_1$$

is safe when x has use 1. Another is that the rule

$$\text{let } x = e_1 \text{ in } \lambda y.\, e_2 \Longrightarrow \lambda y.\, \text{let } x = e_1 \text{ in } e_2$$

is safe when the function $\lambda y.\, e_2$ has use 1. Both of these transformations are used extensively in the Glasgow Haskell Compiler. Until now, their safety was ensured only by ad hoc techniques. And the ad hoc techniques were not adequate – an unsafe version of the second rule was accidentally allowed, with the result that the compiler itself (which is bootstrapped) was slowed by as much as one third [SP95].

Another example of program transformation is the deforestation algorithm [Wad90a]. In order to ensure safety, this algorithm requires that variables be used at most once. The definition of 'used at most once' is easy because deforestation applies to a first-order language. Attempts to apply deforestation to higher-order [MW92] have been hindered by the lack of a suitable definition of 'used at most once' at higher-order. This paper provides such a definition.

1.4 Outline

This paper is organised as follows. Section 2 introduces the language and its semantics. Section 3 describes the fundamentals of the type system. Section 4 discusses principal types and polymorphism. Section 5 summarises the call-by-need reduction rules. Section 6 discusses how to adapt the analysis so that it is appropriate for a call-by-value language. Section 7 describes related work. Section 8 concludes.

$$\text{Var-Once}\ \frac{|x| = 1 \qquad \langle H_1\rangle\, e \Downarrow \langle H_2\rangle\, v}{\langle H_1,\ \text{let } x = e,\ H_3\rangle\, x \Downarrow \langle H_2,\ H_3\rangle\, v} \qquad \text{Var-Many}\ \frac{|x| = \omega \qquad \langle H_1\rangle\, e \Downarrow \langle H_2\rangle\, v}{\langle H_1,\ \text{let } x = e,\ H_3\rangle\, x \Downarrow \langle H_2,\ \text{let } x = v,\ H_3\rangle\, v}$$

$$\text{Var-Rec}\ \frac{}{\langle H_1,\ \text{letrec } x = v,\ H_2\rangle\, x \Downarrow \langle H_1,\ \text{letrec } x = v,\ H_2\rangle\, v}$$

$$\text{Abs}\ \frac{}{\langle H\rangle\, \lambda x.\, e \Downarrow \langle H\rangle\, \lambda x.\, e} \qquad \text{App}\ \frac{\langle H_1\rangle\, e_1 \Downarrow \langle H_2\rangle\, \lambda x.\, e_2 \qquad \langle H_2\rangle\, [y/x]e_2 \Downarrow \langle H_3\rangle\, v}{\langle H_1\rangle\, e_1\, y \Downarrow \langle H_3\rangle\, v}$$

$$\text{Int}\ \frac{}{\langle H\rangle\, n \Downarrow \langle H\rangle\, n} \qquad \text{Plus}\ \frac{\langle H_1\rangle\, e_1 \Downarrow \langle H_2\rangle\, n_1 \qquad \langle H_2\rangle\, e_2 \Downarrow \langle H_3\rangle\, n_2}{\langle H_1\rangle\, e_1 + e_2 \Downarrow \langle H_3\rangle\, \underline{n_1 + n_2}}$$

$$\text{Nil}\ \frac{}{\langle H\rangle\, \text{nil} \Downarrow \langle H\rangle\, \text{nil}} \qquad \text{Cons}\ \frac{}{\langle H\rangle\, \text{cons}\, x\, y \Downarrow \langle H\rangle\, \text{cons}\, x\, y}$$

$$\text{Case-Nil}\ \frac{\langle H_1\rangle\, e_1 \Downarrow \langle H_2\rangle\, \text{nil} \qquad \langle H_2\rangle\, e_2 \Downarrow \langle H_3\rangle\, v}{\langle H_1\rangle\, \text{case } e_1 \text{ of } \{\text{nil} \to e_2;\ \text{cons}\, x\, y \to e_3\} \Downarrow \langle H_3\rangle\, v}$$

$$\text{Case-Cons}\ \frac{\langle H_1\rangle\, e_1 \Downarrow \langle H_2\rangle\, \text{cons}\, y_1\, y_2 \qquad \langle H_2\rangle\, [y_1/x_1, y_2/x_2]e_3 \Downarrow \langle H_3\rangle\, v}{\langle H_1\rangle\, \text{case } e_1 \text{ of } \{\text{nil} \to e_2;\ \text{cons}\, x_1\, x_2 \to e_3\} \Downarrow \langle H_3\rangle\, v}$$

$$\text{Let}\ \frac{\text{fresh } x' \qquad \langle H_1,\ \text{let } x' = e_1\rangle\, [x'/x]e_2 \Downarrow \langle H_2\rangle\, v}{\langle H_1\rangle\, \text{let } x = e_1 \text{ in } e_2 \Downarrow \langle H_2\rangle\, v} \qquad \text{Letrec}\ \frac{\text{fresh } x' \qquad \langle H_1,\ \text{letrec } x' = [x'/x]v\rangle\, [x'/x]e \Downarrow \langle H_2\rangle\, v}{\langle H_1\rangle\, \text{letrec } x = v \text{ in } e \Downarrow \langle H_2\rangle\, v}$$

Figure 1: Natural semantics

2 Language

We now present the syntax and operational semantics of a call-by-need lambda calculus extended with integers, lists, and recursion.

2.1 Terms

The syntax of the language is given below. We use the syntax of Launchbury [Lau93] where arguments in applications and in 'cons' are restricted to variables .

Variables	x, y, z	
Values	$v ::=$	$\lambda x.\, e \mid n \mid \text{nil} \mid \text{cons}\, x\, y$
Terms	$e ::=$	$v \mid x \mid e\, x \mid e_1 + e_2 \mid$
		$\text{case } e_1 \text{ of } \{\text{nil} \to e_2;\ \text{cons}\, x\, y \to e_3\} \mid$
		$\text{let } x = e_1 \text{ in } e_2 \mid \text{letrec } x = v \text{ in } e$

It is trivial to translate terms with the standard syntax for application and 'cons' into the restricted syntax, for example we can translate '$e_1\, e_2$' to 'let $x = e_2$ in $e_1\, x$'. The syntax closely resembles the STG language [Pey92].

Our syntax differs from Launchbury's in three respects. First, we distinguish between non-recursive 'let' and recursive 'letrec' bindings; second, we allow only a single binding in 'letrec', rather than several mutually recursive bindings; and, third, we restrict letrec-bound expressions to be values. The third restriction is required to permit the second, since a single recursive binding such as

$$\text{letrec } y = (\text{let } x = e_1 \text{ in } e_2) \text{ in } e_3,$$

has essentially the same meaning as the following mutually recursive binding

$$\text{letrec } x = e_1 \text{ and } y = e_2 \text{ in } e_3.$$

The restricted 'letrec' is still powerful enough to define recursive functions and cyclic lists. In a companion technical report [MTW95] we show how our usage analysis can be extended to handle mutually-recursive 'letrec' bindings.

2.2 Use annotations

The operational semantics of this section and the reduction rules of Section 5 require that let-bound variables be annotated with uses.

Each let-bound variable x is annotated with a use $|x|$, which is either 1 or ω. If $|x| = 1$, then x may be used at most once during evaluation, and if $|x| = \omega$ then x may be used any number of times. No annotation is required for letrec-bound variables, as they always have use ω. Note that we do not specify how one chooses annotations for variables (in particular, we do *not* require terms to be well-typed).

2.3 Heaps

A *heap* abstracts the state of the store at a point in the computation. It consists of a sequence of bindings associating variables with terms.

Heaps	H	$::=$	$B_1, \ldots, B_n$
Bindings	B	$::=$	$\text{let } x = e \mid \text{letrec } x = v$

We distinguish between non-recursive bindings (written 'let $x = e$') and recursive bindings (written 'letrec $x = v$'). A *configuration* pairs a heap with a term, and is written $\langle H\rangle\, e$.

The expression e in the heap '$H_1, \text{let } x = e, H_2$' can only refer to variables bound in H_1. Similarly, the value v in the heap '$H_1, \text{letrec } x = v, H_2$' can only refer to the recursively-defined variable x and the variables bound in H_1.

2.4 Evaluation rules

Figure 1 presents a natural semantics for lazy evaluation, which closely resembles the one given by Launchbury. The key difference is that the evaluation of a let-bound variable depends on its use annotation.

Evaluation rules have the form $\langle H_1 \rangle\, e \Downarrow \langle H_2 \rangle\, v$, meaning that evaluating expression e in initial heap H_1 returns value v and final heap H_2.

Rule Var-Once evaluates a variable that is used at most once. It looks up the expression e that is bound to the variable x in the heap and then evaluates e. As the variable will no longer be used, its binding is removed from the heap. Note that the expression e in the heap '$H_1, \text{let } x = e, H_3$' can only refer to variables bound in H_1.

Rule Var-Many evaluates a variable that may be used many times. It looks up the expression e that is bound to the variable x in the heap, evaluates e, and updates the heap to bind x to the resulting value. In practice, the update required by Var-Many may have a significant cost, whereas Var-Once avoids this cost.

Rule Var-Rec evaluates a recursively bound variable by simply looking up the value v that is bound to the variable x in the heap. It is simpler because 'letrec' can only bind a value.

Rule Abs is trivial, since a lambda abstraction is already a value. Rule App evaluates applications by evaluating the function to yield a lambda abstraction, then evaluating the body of the abstraction with the argument substituted for the bound variable of the abstraction.

Rules Int and Plus are easy. Rules Nil and Cons are trivial since both 'nil' and '$\text{cons}\, x\, y$' are already values. The two Case rules are again straightforward.

Rules Let and Letrec are similar, each creating a new binding on the heap. In these rules x' is a fresh name not appearing in the expression or the heap.

Observe that all bindings added to the heap use fresh variables, so it trivially follows that all bindings in the heap are unique. (In Launchbury's formulation, renaming occurs during variable access rather than during evaluation of a binding construct.)

3 A use type system

We now present a type system which indicates when values will be accessed at most once. For simplicity, the type system is monomorphic; extensions which allow polymorphism are discussed in Section 4.

3.1 Uses and Constraints

Types will be annotated with *uses*. A type is annotated with use 1 if each value of that type is used at most once, and annotated with ω otherwise. To enable type inference, we also allow use variables (ranged over by j, k, l, m).

$$\text{Uses} \quad \kappa \quad ::= \quad j \mid 1 \mid \omega$$

We use Θ to record the constraints generated by our typing rules. We define Θ to be a set of constraints of the form $j \leq \{k_1, \ldots, k_n\}$. The constraint set Θ is essential if we wish to have principal types for terms, since it allows us to accumulate constraints on a use variable without forcing it to be (prematurely) instantiated to 1 or ω.

The following rules define an ordering on uses, parameterised on a constraint set Θ:

$$\text{Omega}\ \frac{}{\kappa \leq_\Theta \omega} \qquad \text{One}\ \frac{}{1 \leq_\Theta \kappa} \qquad \text{Refl}\ \frac{}{\kappa \leq_\Theta \kappa}$$

$$\text{Taut}\ \frac{(j \leq \{k_1, \ldots, k_n\}) \in \Theta}{j \leq_\Theta k_i}$$

3.2 Types

Types include type variables (let a, b, c range over these), function types, integers, and list types.

$$\text{Types} \quad \tau \quad ::= \quad a^\kappa \mid \tau \rightarrow^\kappa \tau' \mid \text{Int}^\kappa \mid [\tau]^\kappa$$

The type a^κ indicates that the type variable a ranges over types with use κ. The type $\tau \rightarrow^\kappa \tau'$ denotes functions from type τ to type τ' that can be used at most κ times, Int^κ denotes integer values that can be used at most κ times, and $[\tau]^\kappa$ denotes lists with elements of type τ, where the list can be accessed at most κ times.

Write $|\tau|$ for the use attached to type τ, defined as below:

$$|a^\kappa| = \kappa \qquad |\tau \rightarrow^\kappa \tau'| = \kappa \qquad |\text{Int}^\kappa| = \kappa \qquad |[\tau]^\kappa| = \kappa$$

We impose the following well-formedness condition on list types:

The type $[\tau]^\kappa$ is well-formed only if $\kappa \leq_\Theta |\tau|$.

In other words, if a list can be accessed many times, then its elements also might be accessed many times (through the list). A similar restriction appears in the type systems of Guzmán and Hudak [GH90] and Wadler [Wad90b, Wad91].

3.3 Contexts

A context associates a type with each variable that may appear in a term, and is represented by a list of entries of the form $x : \tau$.

$$\text{Contexts} \quad \Gamma, \Delta \quad ::= \quad x_1 : \tau_1, \ldots, x_n : \tau_n$$

Each variable in a context must be distinct. If $x : \tau$ is in Γ, we say that x has use κ if $|\tau| = \kappa$. If Γ and Δ are contexts containing no variables in common, write Γ, Δ to denote the concatenation of the two contexts.

We extend our ordering on uses so that it applies to complete contexts, written $\kappa \leq_\Theta |\Gamma|$ and defined as below:

$$\kappa \leq_\Theta |x_1 : \tau_1, \ldots, x_n : \tau_n| \quad \text{iff} \quad \kappa \leq_\Theta |\tau_i| \text{ for all } i$$

Consider the constraint $\kappa \leq_\Theta |\Gamma|$. If $\kappa = 1$, then no constraint is placed on any use in Γ. But if $\kappa = \omega$ then for every entry $x_i : \tau_i$ in Γ, our definition implies that $|\tau_i| = \omega$.

3.4 Typing judgements

Typing judgements take the form $\Gamma \vdash_\Theta e : \tau$, indicating that in context Γ, and under the constraints Θ, the term e has type τ. The type rules are shown in Figure 2.

The type rules are quite similar to the usual rules for lambda calculus, so we concentrate on explaining the unusual features: the structural rules, and the constraints on uses.

$$\text{Cont}\ \frac{\Gamma,\, x:\tau,\, y:\tau \vdash_\Theta e:\tau' \qquad |\tau| = \omega}{\Gamma,\, z:\tau \vdash_\Theta [z/x, z/y]e:\tau'} \qquad \text{Weak}\ \frac{\Gamma \vdash_\Theta e:\tau'}{\Gamma,\, x:\tau \vdash_\Theta e:\tau'} \qquad \text{Exch}\ \frac{\Gamma,\, x:\tau_1,\, y:\tau_2,\, \Delta \vdash_\Theta e:\tau}{\Gamma,\, y:\tau_2,\, x:\tau_1,\, \Delta \vdash_\Theta e:\tau}$$

$$\text{Var}\ \frac{}{x:\tau \vdash_\Theta x:\tau} \qquad \text{Abs}\ \frac{\Gamma,\, x:\tau \vdash_\Theta e:\tau' \qquad \kappa \leq_\Theta |\Gamma|}{\Gamma \vdash_\Theta \lambda x.\, e:\tau \rightarrow^\kappa \tau'} \qquad \text{App}\ \frac{\Gamma \vdash_\Theta e:\tau \rightarrow^\kappa \tau' \qquad \Delta \vdash_\Theta x:\tau}{\Gamma,\, \Delta \vdash_\Theta e\,x:\tau'}$$

$$\text{Nil}\ \frac{}{\vdash_\Theta \text{nil}:[\tau]^\kappa} \qquad \text{Int}\ \frac{}{\vdash_\Theta n:\text{Int}^\kappa} \qquad \text{Plus}\ \frac{\Gamma \vdash_\Theta e_1:\text{Int}^{\kappa_1} \qquad \Delta \vdash_\Theta e_2:\text{Int}^{\kappa_2}}{\Gamma,\, \Delta \vdash_\Theta e_1 + e_2:\text{Int}^\kappa}$$

$$\text{Cons}\ \frac{\Gamma \vdash_\Theta x:\tau \qquad \Delta \vdash_\Theta y:[\tau]^\kappa}{\Gamma,\, \Delta \vdash_\Theta \text{cons}\,x\,y:[\tau]^\kappa} \qquad \text{Case}\ \frac{\Gamma \vdash_\Theta e_1:[\tau]^\kappa \qquad \Delta \vdash_\Theta e_2:\tau' \qquad \Delta,\, x:\tau,\, y:[\tau]^\kappa \vdash_\Theta e_3:\tau'}{\Gamma,\, \Delta \vdash_\Theta \text{case}\ e_1\ \text{of}\ \{\text{nil} \rightarrow e_2;\ \text{cons}\,x\,y \rightarrow e_3\}:\tau'}$$

$$\text{Let}\ \frac{\Gamma \vdash_\Theta e:\tau \qquad \Delta,\, x:\tau \vdash_\Theta e':\tau'}{\Gamma,\, \Delta \vdash_\Theta \text{let}\ x = e\ \text{in}\ e':\tau'} \qquad \text{Letrec}\ \frac{\Gamma,\, x:\tau \vdash_\Theta v:\tau \qquad \Delta,\, x:\tau \vdash_\Theta e:\tau' \qquad |\tau| = \omega}{\Gamma,\, \Delta \vdash_\Theta \text{letrec}\ x = v\ \text{in}\ e:\tau'}$$

Figure 2: Type rules

3.5 Structural rules

The manipulation of contexts is carefully designed so that if any variable is used more than once this will be indicated by the presence of the structural rule contraction (Cont), which introduces the use ω.

Terms that may be evaluated together are typed in different contexts which are then combined, as can be seen in rules App, Plus, Cons, Case, and Let. As all variables in a context must be distinct, the only way for the same variable to be used more than once is via the Cont rule. In this rule, the substitution $[z/x, z/y]e$ replaces all occurrences of the placeholder variables x and y in term e by the variable z. The type of z (and its placeholders x and y) must be annotated with the usage ω. For instance, here is a type tree for the term $z + z$.

$$\cfrac{\cfrac{\cfrac{}{x:\text{Int}^\omega \vdash_\Theta x:\text{Int}^\omega}\,\text{Var} \qquad \cfrac{}{y:\text{Int}^\omega \vdash_\Theta y:\text{Int}^\omega}\,\text{Var}}{x:\text{Int}^\omega,\, y:\text{Int}^\omega \vdash_\Theta x + y:\text{Int}^j}\,\text{Plus}}{z:\text{Int}^\omega \vdash_\Theta z + z:\text{Int}^j}\,\text{Cont}$$

As one would expect, the variable z has use ω. The use variable j on the result type may be instantiated to 1 or ω, depending on how the result of the addition is used.

If a variable is never used, this is indicated by the presence of the structural rule weakening (Weak). This rule places no constraints on the use, since the use 1 (at most once) and the use ω (any number of times) are both compatible with not being used at all. However, the weakening rule may be helpful in devising a type system for strictness analysis, and is certainly important in usage analysis for call-by-value languages (see Section 6).

The last structural rule, exchange (Exch), simply indicates that the order of bindings in a context is irrelevant.

The contraction, weakening, and exchange rules are not syntax directed, but do not pose an impediment to the existence of principal types since it easy to devise an algorithm which determines whether contraction or weakening must be used on each variable, placing these rules as close to the root of the type tree as possible.

A subtlety in the manipulation of contexts is revealed by the Case rule. In the Case rule, the term e_1 is always evaluated, and then either e_2 or e_3 is evaluated. Hence it makes sense to type e_1 in a different context from e_2 and e_3, but to type e_2 and e_3 in the same context. For instance, the following is a valid typing.

$$\begin{array}{l} x:\text{Int}^1,\, xs:[\text{Int}^1]^1 \vdash_\Theta \\ \quad \text{case}\ xs\ \text{of}\ \{\text{nil} \rightarrow x;\ \text{cons}\,y\,ys \rightarrow x + y\}:\text{Int}^1 \end{array}$$

Although x appears twice in the term, it is only labelled as being used once, which is correct because only one branch of the 'case' term will ever be evaluated.

3.6 Term rules

In rule Abs, the constraint $\kappa \leq_\Theta |\Gamma|$ reflects the fact that if a function abstraction may be accessed more than once, then every free variable of that abstraction may be accessed more than once.

Consider again Example 3 from the introduction.

$$\begin{array}{l} \text{let}\ x = 1 + 2\ \text{in} \\ \quad \text{let}\ f = \lambda z.\, x + z\ \text{in} \\ \quad\quad f\,3 + f\,4 \end{array}$$

Since f appears twice in $f\,3 + f\,4$ it has use ω. Since x is a free variable of a lambda abstraction with use ω, it is in turn forced to have use ω. Thus, despite appearing only once in the term, x must be labelled with use ω, as indeed it should be since it will be accessed twice in the course of evaluation.

In the following example, we do not know how many times the abstraction $\lambda z.\, x + z$ will be used (since such usage information is, in general, a property of the enclosing program). Fortunately, we can give $\lambda z.\, x + z$ usage j, where j is a use variable which can be instantiated to either 1 or ω at some later point in the analysis.

$$x:\text{Int}^k \vdash_\Theta \lambda z.\, x + z:\text{Int}^1 \rightarrow^j \text{Int}^1, \quad \Theta = \{j \leq \{k\}\}$$

In the above example, the usage variables j and k are not independent, since setting j to ω should force k to be ω (x will certainly be used many times if the abstraction is). The constraint set Θ records this dependency, allowing us to satisfy the constraint $\kappa \leq_\Theta |\Gamma|$ in the Abs rule without (prematurely) instantiating j and k.

Consider the following example, where the term '$\text{cons}\,x\,y$' is used twice.

$$\begin{array}{l} \text{let}\ l = \text{cons}\,x\,y\ \text{in} \\ \quad \text{case}\ l\ \text{of}\ \cdots\ \text{case}\ l\ \text{of}\ \cdots \end{array}$$

Our typing rules give 'cons $x\,y$' the type $[a^{\omega}]^{\omega}$ which in turn forces x to have type a^{ω} and y to have type $[a^{\omega}]^{\omega}$ as expected, since both x and y may be accessed twice. Note that the Nil and Cons rules implicitly include the condition $\kappa \leq_{\Theta} |\tau|$ because of our global well-formedness condition on list types (see Section 3.2).

The Plus rule deserves some explanation. Our addition operator is strict, so the result of evaluating $e_1 + e_2$ will simply be an integer constant which will not refer to any part of the results of evaluating e_1 and e_2. Therefore, the usage assigned to the expression $e_1 + e_2$ need not depend at all on the usages κ_1 and κ_2. A similar argument applies to the App rule, since application is strict in its first argument.

3.7 Recursion

In a recursive definition, even a single access to a variable may allow additional accesses via the recursion. The Letrec rule therefore forces every letrec-bound variable to have use ω. This does not mean that whenever recursion is involved all uses must degenerate to ω. If a function is defined recursively, the argument and result of the function may still have use 1. For example, here is a function to append two lists.

```
letrec $append = \lambda xs.\, \lambda zs.$
  case $xs$ of {
    nil $\rightarrow zs$
    cons $y\ ys \rightarrow$ let $as = append\ ys\ zs$ in cons $y\ as$}
in $\cdots$
```

It has the following (principal) type

$$[a^j]^k \rightarrow^{\omega} [a^j]^l \rightarrow^m [a^j]^l, \qquad k \leq \{j\},\ l \leq \{j\},\ m \leq \{k\}$$

The constraints $k \leq \{j\}$ and $l \leq \{j\}$ are generated by our global well-formedness condition on list types, and indicate that if the argument or result lists are accessed more than once, then the elements of those lists may also be accessed more than once. The constraint $m \leq \{k\}$ is generated by the Abs rule, and indicates that if *append* is partially applied and then used more than once, the first argument list may be accessed more than once.

One instance of the above type is

$$[\mathrm{Int}^{\omega}]^{\omega} \rightarrow^{\omega} [\mathrm{Int}^{\omega}]^{\omega} \rightarrow^{\omega} [\mathrm{Int}^{\omega}]^{\omega}$$

indicating that *append* can take two lists to which there may be multiple pointers, and return a list to which there may be multiple pointers. Another instance is

$$[\mathrm{Int}^1]^1 \rightarrow^{\omega} [\mathrm{Int}^1]^1 \rightarrow^1 [\mathrm{Int}^1]^1$$

indicating that *append* may be applied multiple times to two lists to each of which there is only one pointer, returning a list to which there is only one pointer. (Attaching the use 1 to the second arrow guarantees that one cannot create extra pointers to the first argument list by creating and duplicating a partial application.) For this version of *append* it is possible to generate code that reuses the 'cons' cells of the first argument in producing the result.

3.8 Typeability

The ordinary rules for simply typed lambda calculus can be derived by simply omitting all use annotations and use constraints from the rules given here. It follows that if a term is typeable in this system, it is typeable in simply typed lambda calculus. Conversely, if a term is typeable in simply typed lambda calculus, then it is also typeable in this system (just take all uses to be ω).

4 Principal types and polymorphism

Before discussing what it means for a type to be principal for a given term, we first need to define when a type is an instance of another type. Our definition of instantiation is closely related to Mitchell's definition of instantiation for a type system with simple subtypes [Mit84, Mit91].

4.1 Instantiation

A substitution is a pair of finite maps. One component maps type variables to types, while the other maps use variables to uses. Whenever a type variable is replaced by a type, the new type must have the same usage: for each $(a_i^{\kappa_i} \mapsto \tau_i) \in S$ we require that $S(\kappa_i) = |\tau_i|$.

Type substitutions	$TS ::= \{a_1^{\kappa_1} \mapsto \tau_1, \ldots, a_n^{\kappa_n} \mapsto \tau_n\}$	
Use substitutions	$US ::= \{k_1 \mapsto \kappa_1, \ldots, k_n \mapsto \kappa_n\}$	
Substitutions	$S ::= (TS, US)$	

We can derive an instance of the typing $\Gamma \vdash_{\Theta} e : \tau$ by applying a substitution S to Γ and τ, and replacing Θ with a stronger constraint set Θ'. The behaviour of S on types and contexts is defined in the usual way. We define the conditions under which Θ' is stronger than Θ (under the substitution S) below.

$\Theta' \models S\Theta$ iff for each $(j \leq \{k_1, \ldots, k_n\}) \in \Theta$ we have that $S(j) \leq_{\Theta'} S(k_i)$ for each i.

A straightforward induction on the structure of typing derivations proves the following substitution lemma.

Lemma 4.1.1 (Type substitution)
If $\Gamma \vdash_{\Theta} e : \tau$ and $\Theta' \models S\Theta$ then $S\Gamma \vdash_{\Theta'} e : S\tau$.

4.2 Unification

It is easy to modify Robinson's unification algorithm [Rob65] so that it unifies types containing usage information. However, whenever we unify a usage variable with another usage, we need to update the current constraint set. Suppose Θ is the following constraint set:

$$j \leq \{l\},\ k \leq \{m\},\ l \leq \{m\},\ m \leq \{\}$$

If we unify the types a^j and Int^k we get the substitution:

$$(\{a^j \mapsto \mathrm{Int}^k\}, \{j \mapsto k\})$$

Since we have unified j and k, we must modify the constraint set Θ so that it merges the constraints for j and k:

$$k \leq \{l, m\},\ l \leq \{m\},\ m \leq \{\}$$

Alternatively, if we unify the types a^j and Int^{ω} we get the substitution:

$$(\{a^j \mapsto \mathrm{Int}^{\omega}\}, \{j \mapsto \omega\})$$

We have instantiated j to ω, which in turn forces us to also instantiate l and m to ω. The constraint $k \leq \{m\}$ then simplifies to $k \leq \{\omega\}$, which can be eliminated, since ω is the maximal usage.

If we unify usage variables with other usage variables, or with ω, we can always derive a new constraint set, as explained above. We can only fail to produce a new constraint set if we unify use variables with 1 (for instance, we might unify j with ω and l with 1, generating the unsatisfiable constraint $\omega \leq 1$). Fortunately, it is easy to show that during type inference we never generate such constraints.

4.3 Principal types

Every typeable term e has a principal type judgement, of which all other type judgements for e are instances. The result is proved, as usual, by exhibiting an algorithm that computes principal types.

Proposition 4.3.1 (Principal types)
If $\Gamma \vdash_\Theta e : \tau$ then there exist Γ', Θ' and τ' such that $\Gamma' \vdash_{\Theta'} e : \tau'$ and for all Γ'', Θ'' and τ' such that $\Gamma'' \vdash_{\Theta''} e : \tau''$ there exists a substitution S such that $S\Gamma' \subseteq \Gamma''$, $S\tau' = \tau''$ and $\Theta'' \models S\Theta'$.

4.4 Inferring usage annotations

The operational semantics of Section 2 and the reduction rules of Section 5 require that each let-bound variable x is annotated with a use $|x|$ which is either 1 or ω. Such annotations may be inferred as follows. First, determine a principal typing for the given term, and a corresponding principal type derivation. One component of the principal typing is a constraint set Θ, which may be instantiated using any substitution S such that $\{\} \models S\Theta$. Naturally, we choose the substitution that maps each use variable to 1, since this yields the best usage information. One can now extract the relevant usage information from the (instantiated) type derivation.

4.5 Polymorphism

The next step is to use 'let' terms to introduce polymorphism in the usual way. There are two possibilities. The first is to allow polymorphism only on type variables. For instance, the polymorphic type for *append* would be

$$\forall a^j.\, [a^j]^k \rightarrow^\omega [a^j]^l \rightarrow^m [a^j]^l$$

with the same constraints as before. This allows *append* to be used on lists of different types, but every occurrence of *append* in the program must have the *same* usage labelling. Although crude, an analysis of this sort may be suitable for some purposes, such as update analysis. An existing update analyser, based on abstract interpretation, has a similar limitation but has proved reasonably effective [Mar93].

The second possibility is to allow polymorphism on both type and use variables. For instance, the polymorphic type for *append* would be

$$\begin{gathered}\forall j.\, \forall k \leq \{j\}.\, \forall l \leq \{j\}.\, \forall m \leq \{k\}.\\ \forall a^j.\, [a^j]^k \rightarrow^\omega [a^j]^l \rightarrow^m [a^j]^l\end{gathered}$$

In order to maximise the potential for optimisation, the compiler needs to generate different versions of *append* for different instantiations of the use variables. A similar technique is used in Haskell compilers to specialise code involving overloaded functions, and experience to date suggests that this is feasible and does not necessarily lead to an explosion in code size [Aug93, Jon93]. In some situations, instead of specialising *append* for different uses, we might consider having just one version of *append*, and interpret the use variables j, k, l, m as additional arguments to the *append* function, enabling run-time selection of the behaviour of *append*.

The trade-off between these two possibilities is similar to the trade-off between monovariant and polyvariant binding time analysis in partial evaluation. Further experimentation will be necessary to better understand the strengths and weaknesses of each approach.

5 Reduction and subject-reduction

We previously described the semantics of our language using Launchbury's operational semantics of call-by-need [Lau93]. We now give an alternative characterisation of that semantics using a modification of the call-by-need calculus of Ariola *et al.* [AFMOW95].

Working in the framework of a calculus with reduction rules simplifies our proof of subject-reduction, but more importantly, gives a set of rules which can be used by a compiler to optimise programs without danger of duplicating work (or returning the wrong result). We show how our usage information enables additional "safe" reduction rules to be formulated, allowing more aggressive optimisation when values are known to be used at most once.

5.1 Reduction rules

In the call-by-need calculus of Ariola *et al*, a closure is created for the argument of each function application, whereas in Launchbury's operational semantics only 'let' creates closures. This difference is significant: it means that the model of Ariola *et al.* may create many more closures than the model of Launchbury. For example, consider the following recursive function.

$$\begin{array}{l}\text{letrec } f = \lambda l.\, \lambda y.\\ \quad \text{case } l \text{ of } \{\text{nil} \rightarrow y;\ \text{cons}\, x\, xs \rightarrow f\, xs\, y\}\\ \text{in let } l = e_0 \text{ in let } y = e_1 \text{ in } f\, l\, y\end{array}$$

Launchbury's model does *not* create closures for the arguments (xs and y) to the recursive call of f. The model of Ariola *et al.* creates two (redundant) closures for these arguments during every recursive call of f.

Fortunately, it is straightforward to adapt the calculus of Ariola *et al.* so that it corresponds to the model of Launchbury. The required contextual forms are given below, and the reduction rules are given in Figure 3.

The first group of reduction rules are straightforward. Note that the reduction rules of Ariola *et al.* would reduce the application '$(\lambda x.\, e)\ y$' to 'let $x = y$ in e', but our syntactic restriction on application (which forces all arguments to be variables) means that we can easily avoid creating the (redundant) closure for x, and substitute y for x instead.

The second group of rules are a generalisation of the let-floating and garbage collection rules of Ariola *et al.*, which are necessary to guarantee that every closed term can be reduced to a weak head normal form. The Let contexts, defined below, enumerate those positions from which a let-expression can float outwards.

$$\begin{array}{ll}\text{Let contexts} & L ::= [\,]\ x \mid \text{let } x = [\,] \text{ in } e \mid [\,] + e \mid e + [\,] \mid \\ & \qquad \text{case } [\,] \text{ of } \{\text{nil} \rightarrow e_1;\ \text{cons}\, x\, y \rightarrow e_2\}\end{array}$$

A context C is a term with a hole. Note that the hole cannot appear in the argument of an application or 'cons', since these positions can only contain variables and cannot be replaced by arbitrary terms.

$$\begin{array}{ll}\text{Contexts} & C ::= [\,] \mid C\ x \mid \text{let } x = C \text{ in } e \mid \text{let } x = e \text{ in } C \mid \\ & \qquad \text{letrec } x = C \text{ in } e \mid \text{letrec } x = v \text{ in } C \mid \\ & \qquad \text{case } C \text{ of } \{\text{nil} \rightarrow e_1;\ \text{cons}\, x\, y \rightarrow e_2\} \mid \\ & \qquad C + e \mid e + C\end{array}$$

The key change from the work of Ariola *et al.* involves the rules that allow substitution of a value.

$$\begin{array}{lcl} n_1 + n_2 & \Longrightarrow & \underline{n_1 + n_2} \\ (\lambda x.\, e)\, y & \Longrightarrow & [y/x]e \end{array} \qquad \begin{array}{lcl} \text{case nil of } \{\text{nil} \to e_1;\ \text{cons}\, x\, y \to e_2\} & \Longrightarrow & e_1 \\ \text{case } (\text{cons}\, y_0\, y_1) \text{ of } \{\text{nil} \to e_1;\ \text{cons}\, x_0\, x_1 \to e_2\} & \Longrightarrow & [y_0/x_0, y_1/x_1]e_2 \end{array}$$

$$\begin{array}{lcl} L[\text{let } x = e' \text{ in } e] & \Longrightarrow & \text{let } x = e' \text{ in } L[e] \\ L[\text{letrec } x = v \text{ in } e] & \Longrightarrow & \text{letrec } x = v \text{ in } L[e] \end{array} \qquad \begin{array}{lcll} \text{let } x = e' \text{ in } e & \Longrightarrow & e & \text{if } x \notin fv(e) \\ \text{letrec } x = v \text{ in } e & \Longrightarrow & e & \text{if } x \notin fv(e) \end{array}$$

$$\begin{array}{lcll} \text{letrec } x = v \text{ in } C[x] & \Longrightarrow & \text{letrec } x = v \text{ in } C[v] & \\ \text{let } x = v \text{ in } C[x] & \Longrightarrow & \text{let } x = v \text{ in } C[v] & \text{if } |x| = \omega \\ \text{let } x = v \text{ in } C[x] & \Longrightarrow & C[v] & \text{if } |x| = 1 \end{array}$$

Figure 3: Call-by-need reductions

$$\text{let } x = v \text{ in } C[x] \Longrightarrow \text{let } x = v \text{ in } C[v], \quad \text{if } |x| = \omega$$

The above rule is safe because, since v is already a value, the substitution cannot duplicate computation. However, in order to guarantee that reductions preserve use types, we must restrict this rule to the case where all free variables of v have use ω; otherwise we may duplicate a variable with use 1 resulting in an ill-typed term. An adequate restriction is to require that x has use ω, since this implies that all free variables of v also have use ω, as the reader may easily check (the two important cases are when v is a function abstraction and when v is a 'cons').

In the case where x has use 1, we must eliminate the let-expression as soon as we substitute for x (this reduction corresponds to the Var-Once rule in the operational semantics, which avoids updating closures for variables which are used at most once).

$$\text{let } x = v \text{ in } C[x] \Longrightarrow C[v], \quad \text{if } |x| = 1$$

We take the compatible closure of the rules shown in Figure 3 as our reduction relation. Capture of free variables is disallowed, so context C should not bind x in the three rules containing C, and context L should not have x as a free variable in the two rules containing L.

5.2 Soundness and completeness

If H is a heap, as used in the operational semantics of Section 2, and e is a term then the configuration $\langle H \rangle\, e$ corresponds to the term 'let $\langle H \rangle$ in e', defined as follows.

$$\begin{array}{ll} \text{let } \langle\ \rangle \text{ in } e & = e \\ \text{let } \langle \text{let } x = e', H \rangle \text{ in } e & = \text{let } x = e' \text{ in } (\text{let } \langle H \rangle \text{ in } e) \\ \text{let } \langle \text{letrec } x = v, H \rangle \text{ in } e & = \text{letrec } x = v \text{ in } (\text{let } \langle H \rangle \text{ in } e) \end{array}$$

We can now prove that the reduction rules are sound and complete with respect to the operational semantics. (We have not checked the soundness proof in detail, but believe that it should be a straightforward adaption of the soundness result proved by Ariola *et al.*)

Conjecture 5.2.1 (Soundness)
If let $\langle H \rangle$ in $e \Longrightarrow^*$ let $\langle H'' \rangle$ in v'' *then there exist* H' *and* v' *such that* $\langle H \rangle\, e \Downarrow \langle H' \rangle\, v'$ *and* let $\langle H' \rangle$ in $v' \Longrightarrow^*$ let $\langle H'' \rangle$ in v''.

Proposition 5.2.2 (Completeness)
If $\langle H \rangle\, e \Downarrow \langle H' \rangle\, v'$ *then* let $\langle H \rangle$ in $e \Longrightarrow^*$ let $\langle H' \rangle$ in v'

The soundness and completeness results take an even simpler form for terms of type integer.

Corollary 5.2.3 (Soundness and completeness)
There exists a heap H' *such that* $\langle H \rangle\, e \Downarrow \langle H' \rangle\, n$ *if and only if* let $\langle H \rangle$ in $e \Longrightarrow^* n$.

5.3 Subject Reduction

Use types are preserved by reduction. The proof is straightforward, verifying each rule in Figure 3 separately, and using structural induction over terms for the compatible closure.

Proposition 5.3.1 (Subject reduction)
If $\Gamma \vdash_\Theta e : \tau$ *and* $e \Longrightarrow e'$ *then* $\Gamma \vdash_\Theta e' : \tau$.

Combining Propositions 5.3.1 and 5.2.2 yields a soundness result for our type system with respect to the operational semantics.

Corollary 5.3.2 (Operational subject reduction)
If $\vdash_\Theta$ let $\langle H \rangle$ in $e : \tau$ *and* $\langle H \rangle\, e \Downarrow \langle H' \rangle\, v$ *then* $\vdash_\Theta$ let $\langle H' \rangle$ in $v : \tau$.

5.4 Additional transformations

There are many useful program transformation rules that we might add to those appearing in Figure 3.

For instance, it is helpful to have the reduction

$$\begin{array}{l} \text{let } x = e_1 \text{ in } (\text{let } y = e_2 \text{ in } e_3) \Longrightarrow \\ \text{let } y = (\text{let } x = e_1 \text{ in } e_2) \text{ in } e_3, \qquad \text{if } x \notin fv(e_3), \end{array}$$

which avoids the creation of a closure for e_1 if, say, y is not evaluated at run time. By the way, note that the rule $L[\text{let } x = e' \text{ in } e] \Longrightarrow \text{let } x = e' \text{ in } L[e]$ has as an instance

$$\begin{array}{l} \text{let } y = (\text{let } x = e_1 \text{ in } e_2) \text{ in } e_3 \Longrightarrow \\ \text{let } x = e_1 \text{ in } (\text{let } y = e_2 \text{ in } e_3), \qquad \text{if } x \notin fv(e_3), \end{array}$$

which is helpful if, say, further simplification reduces e_2 to a value. The Glasgow Haskell compiler makes extensive use of both reductions [PS94]. The fact that the latter rule undoes the effect of the former is not a problem in practice, since one can apply the rules in separate phases: first flatten out as many 'let's as possible using the second rule, do any transformations which were enabled by the flattening phase, then finally push any remaining 'let's back inwards using the first rule.

We could, in fact, continue and write a number of additional rules which can float let-bound expressions inwards towards their uses. Such transformations complement the substitution rules already present in the reduction rules, since they allow arbitrary let-bound expressions to move inwards (rather than just values, as is the case in the reduction rules).

Most such rules can be formulated purely syntactically (for example, using conditions on free variables such as the condition $x \notin fv(e_3)$ above). However, we require usage information to safely control the following reduction rule, which moves a 'let' inside a lambda abstraction.

$$\text{let } x = e' \text{ in } (\lambda y.\, e) \Longrightarrow \lambda y.\, (\text{let } x = e' \text{ in } e), \quad \text{if } |\lambda y.\, e| = 1$$

We need to add a further use annotation to terms: we assume that each abstraction $\lambda x.\, e$ is annotated with a use of 1 or ω which we write $|\lambda x.\, e|$. The use information is crucial for safety: the above reduction might duplicate computation if $|\lambda y.\, e| = \omega$ (the expression '$\lambda y.$ (let $x = e'$ in e)' evaluates e' every time it is applied, while 'let $x = e'$ in $(\lambda y.\, e)$' evaluates e' at most once). Again, this reduction is extensively used in the Glasgow Haskell compiler. It turns out to be particularly important for one form of deforestation [GLP93].

Let-floating rules such as those described above can be used to achieve the effect of "safe" expression inlining. Moreover, the let-floating rules all preserve the type of a term in our system, enabling efficient type-based optimisation (since usage information need not be recalculated during optimisation).

6 Use analysis for call-by-value calculi

Our use analysis can easily be adapted for call-by-value calculi. In such calculi, we are interested in variables which are used *exactly* once (rather than *at most* once). For example, if f is used exactly once, and x is only used in the body of f, then we can safely transform

$$\text{let } x = e \text{ in let } f = \lambda y.\, x + 3 \text{ in } \ldots$$

into

$$\text{let } f = \lambda y.\, e + 3 \text{ in } \ldots$$

(hopefully reducing the maximum amount of storage used by the program, and potentially exposing other optimisations). This transformation is clearly unsafe if f is never used, since the expression e might be non-terminating — we would have transformed a non-terminating program into a terminating program.

Changing our type system to determine when a value is used *exactly* once is easy. We simply change the weakening rule as below:

$$\text{Weak}\ \frac{\Gamma \vdash_\Theta e : \tau' \qquad |\tau| = \omega}{\Gamma,\, x : \tau \vdash_\Theta e : \tau'}$$

We can now interpret the use 1 as meaning that a value is used *exactly* once. This transforms our type system into something much closer to a *linear* type system. In a companion paper [MOTW95], we elaborate on the connections between linear logic and call-by-value reduction, and affine logic and call-by-need reduction.

We conjecture that usage-based program transformation in the presence of side-effects could be handled by combining our usage analysis with an effect system [Luc87, LG88, JG91]. (Effect systems can be used to distinguish side-effecting computation from purely functional computation.)

7 Related work

7.1 Linear logic

The type system presented here is based on ideas taken from the linear logic of Girard [Gir87] and its successor, the Logic of Unity [Gir93]. A companion paper describes the embedding of the call-by-need calculus into linear logic that underlies the type system used here [MOTW95]. Interested readers are referred to that paper for a survey of related work on linear logic.

Inspired by Girard's Logic of Unity, we might be tempted to refine our type system by annotating variables separately from types, thus allowing a variable to be annotated with 1 even if its type has use ω. For example, in the following expression our type system assigns *both* x and y use ω (since x and y are forced to have the same type).

$$\begin{array}{c}\text{let } x = 1 + 2 \text{ in} \\ \text{let } y = x \text{ in} \\ \cdots x \cdots y \cdots\end{array}$$

Examples such as the one above are relatively common in practice, but do not cause any problems since trivial let-bindings such as 'let $y = x$ in $\cdots$' can always be eliminated during optimisation. It is possible that there exist other inaccuracies which cannot be circumvented so easily, but we choose to retain our simpler analysis until we have implemented it in the Glasgow Haskell compiler and measured the effect (if any) of such refinements on real-life programs.

7.2 Call-by-need analyses

We are aware of two other analyses that attempt to determine when a value is used at most once under call-by-need evaluation. One is a type system due to Launchbury and others [Lau92], the other is an abstract interpretation due to Marlow [Mar93]. We note three points of comparison.

First, unlike ours, neither of the other analyses possess a proof of soundness. Second, our system sometimes derives more precise information than the other two; see Example 3 in Section 1.1. Third, unlike the above analyses, our type system does not detect the case where closures are *never* used (we omitted the zero usage from our analysis so as to simplify our usage constraints).

Our next step is to implement our analysis in the Glasgow Haskell compiler, allowing us to compare it directly with Marlow's. By observing how our analysis performs on real programs we can test whether omitting zero usages has a significantly impact.

Concurrent Clean's *uniqueness types* [BS93] are related to our call-by-need analysis since they are proved sound with respect to a graph-reduction semantics which correctly models the sharing of reductions that occur in an implementation. However, uniqueness types are designed to deal with data structure update, rather than program transformation and closure update (see Section 7.4 for a more detailed comparison).

7.3 Call-by-name analyses

We also are aware of two analyses that determine usage information for values under call-by-name reduction, one due to Wright and Baker-Finch [WB93], the other due to Courtenage and Clack [CC94]. Both are based on type systems, and both have been argued to be sound. We note three points of comparison.

First, choosing call-by-name evaluation instead of call-by-need prevents even fairly simple optimisations from being discovered. For instance, in Example 1 of Section 1.1, the variable x is used once under call-by-need, as our system discovers, but twice under call-by-name. Experience suggests this difference is significant, as the situation encountered in Example 1 is fairly common.

Second, even if we are satisfied to limit our attention to call-by-name, neither system provides an especially useful analysis. The Wright and Baker-Finch system discovers too much information: function types are annotated with natural numbers which indicate the number of times a function uses its argument, and this level of accuracy renders the type system undecidable. The Courtenage and Clack system discovers too little information: they can determine when an argument is used zero times, exactly once, or at least once; but (because of the lack of disjunctive types) they cannot determine when an argument is used at most once, which is most helpful for the problems we are interested in.

Third, unlike our system, the other systems also provide information about when a value is used at least once, which is useful for strictness analysis. For this question, the distinction between call-by-need and call-by-name is irrelevant, which may explain why the authors of these systems were willing to settle for a call-by-name analysis.

7.4 Data structure update

A number of analyses for in-place update of data structures have been proposed, including those by Schmidt [Sch85], Hudak [Hud86], Baker [Bak90], Guzmán and Hudak [GH90], Wadler [Wad90b, Wad91], and Barendsen and Smetsers [BS93]. In such analyses it is sufficient to check that there can be at most one *live* pointer to a data structure.

For example, in the following expression the 'then' and 'else' clauses can only be executed after the predicate $p(x)$ has evaluated to a boolean. But a boolean cannot contain a reference to x, so the second reference to x in the 'then' clause is never live at the same time as the reference to x in the predicate $p(x)$.

$$\text{let } x = e \text{ in } (\text{if } p(x) \text{ then } x \text{ else } y)$$

This refined usage information, which is not discovered by our analysis, is extremely useful when considering in-place update of data structures. However, inspection of the above example also reveals that such information is of little use for program transformation or update analysis: even though x has at most one live reference, we *cannot* substitute e for x without duplicating work, nor can we avoid updating the closure for x.

8 Conclusions

We have presented a simple type system which can determine when a value is used at most once, even in the presence of higher-order functions and data structures. Our analysis is tailored to the precise reduction strategy used in the Glasgow Haskell compiler, and therefore yields more accurate results than analyses which assume call-by-name reduction. We have proved our type system sound with respect to Launchbury's natural semantics of lazy evaluation, and have provided safe reduction rules which the compiler can use to transform programs without risking duplicating work.

A prototype type inference algorithm has already been implemented. Our next step is to incorporate our type system into the Glasgow Haskell compiler [PHHPW93]. This will enable us to measure the effect of our optimisations on large Haskell programs. The Glasgow Haskell compiler uses an explicitly-typed core language to express most of its program transformations. By adding our usage information to the core language type system we can conveniently provide information to the optimiser, enabling additional program transformations, and allowing the code generator to omit unnecessary closure updates.

Our annotated types also provide a convenient way of communicating usage information across module boundaries (we simply add usage information to the user-level type information which is already exported from a module).

We intend to further explore how our type system enables in-place update of data structures. An interesting question is how much of this should be done automatically by the compiler, how much should be under the control of the user, and to what extent the type system acts as an effective mechanism to let the user understand and control optimisations.

The information computed by our analysis seems at first glance to be similar to that which is required for *strictness analysis*. However, for the purposes of strictness analysis one needs to know when an expression is used *at least* once. We conjecture that a generalisation of our type system might be able to provide simple strictness information at the same time as inferring information for use in program transformation and update analysis.

9 Acknowledgements

The ideas described here grew out of collaboration with the Glasgow functional programming group, especially Simon Peyton Jones, John Launchbury, Andy Gill, Simon Marlow, Will Partain, Patrick Sansom, and Andre Santos. The Semantique II working group provided helpful feedback, especially Patrick Cousot, Sebastian Hunt, Torben Mogensen, Dave Sands, and Yan-Mei Tang.

References

[AFMOW95] Z. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. *Symposium on Principles of Programming Languages*, ACM Press, San Francisco, California, January 1995.

[Aug93] L. Augustsson, Implementing Haskell overloading. In *Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[Bak90] H. Baker, Unify and conquer (garbage, updating, aliasing ...) in functional languages. In *Conference on Lisp and Functional Programming*, ACM Press, Nice, June 1990.

[BS93] E. Barendsen, and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems (extended abstract). In *Proceedings 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag LNCS 761, 1993.

[CC94] S. A. Courtenage and C. D. Clack, Analysing resource use in the λ-calculus by type inference, In *ACM*

Sigplan Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1994.

[DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings 9'th ACM Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.

[Gir87] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir93] J.-Y. Girard, On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GH90] J. Guzmán and P. Hudak, Single-threaded polymorphic lambda calculus. In *Proceedings 5'th IEEE Symposium on Logic in Computer Science*, Philadelphia, Pa., June 1990.

[GLP93] A. Gill, J. Launchbury, and S. Peyton Jones, A short-cut to deforestation. In *Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, 146:29–60, December 1969.

[Hud86] P. Hudak, A semantic model of reference counting and its abstraction. In *Proceedings ACM Conference on Lisp and Functional Programming*, Cambridge, Mass., August 1986.

[Jon93] M. Jones, Partial evaluation for dictionary-free overloading. Technical report TR-959, Computer Science Department, Yale University, April 1993.

[JG91] Pierre Jouvelot and D.K. Gifford, Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, 1991.

[Lau92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. Peyton Jones, and P. Wadler. Avoiding unnecessary updates. In *Glasgow Workshop on Functional Programming*, Ayr, July 1992. Springer Verlag Workshops in Computing Series.

[Lau93] J. Launchbury, A natural semantics for lazy evaluation. In *Proceedings 20'th ACM Symposium on Principles of Programming Languages*, Charleston, S.C., January 1993.

[Luc87] J.M. Lucassen, Types and effects, towards an integration of functional and imperative programming. PhD thesis, MIT Laboratory for Computer Science, 1987.

[LG88] J.M. Lucassen and D.K. Gifford, Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, 1988.

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

[Mit84] J. C. Mitchell, Coercion and type inference (summary). In *Proceedings 11'th ACM Symposium on Principles of Programming Languages*, January 1984.

[Mit91] J. C. Mitchell, Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991.

[MOTW95] J. Maraist, M. Odersky, D. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11'th International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana, April 1995.

[MTW95] C. Mossin, D.N. Turner, and P. Wadler, Once Upon A Type. Technical report TR-1995-8, Computing Science Department, University of Glasgow, April 1995.

[MW92] S. Marlow and P. Wadler, Deforestation for higher order functions. In *Glasgow Workshop on Functional Programming*, Ayr, July 1992. Springer Verlag Workshops in Computing Series.

[Pey92] S. Peyton Jones, Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PHHPW93] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, and P.L. Wadler, The Glasgow Haskell compiler: a technical overview. In *Joint Framework for Information Technology (JFIT), Technical Conference Digest*, March, 1993.

[PS94] S. Peyton Jones and A. Santos, Compilation by Transformation in the Glasgow Haskell Compiler. In *Glasgow Workshop on Functional Programming*, Ayr, September 1994. Springer Verlag Workshops in Computing Series.

[Rob65] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1), January 1965.

[Sch85] D. A. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, 7:299–310, 1985.

[SP95] P. Sansom and S. L. Peyton Jones, Time and space profiling for non-strict higher-order functional languages. In *22'nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[Wad90a] P. Wadler, Deforestation: Transforming programs to eliminate trees, *Theoretical Computer Science*, 73, 1990.

[Wad90b] P. Wadler, Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland, 1990.

[Wad91] P. Wadler, Is there a use for linear logic? In *Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, ACM Press, New Haven, Connecticut, June 1991.

[WB93] D. A. Wright and C. A. Baker-Finch, Usage analysis with natural reduction types. In *Workshop on Semantic Analysis*, 1993.