A Usage Analysis with Bounded Usage Polymorphism and Subtyping

Jörgen Gustavsson and Josef Svenningsson

Chalmers University of Technology and Göteborg University

Abstract

Usage analysis aims to predict the number of times a heap allocated closure is used. Previously proposed usage analyses have proved not to scale up well to large programs. In this paper we present a powerful and accurate type based analysis designed to scale up for large programs. The key features of the type system are usage subtyping and bounded usage polymorphism. Bounded polymorphism can lead to huge constraint sets so to express constraints compactly we introduce a new expressive form of constraints which allows constraints to be represented compactly through calls to *constraint abstractions*.

1 Introduction

In the implementation of a lazy functional language sharing of evaluation is performed by updating. For example, the (unoptimised) evaluation of

$$(\lambda x.x + x)\ (1 + 2)$$

proceeds as follows. First, a closure for $1 + 2$ is built in the heap and a reference to the closure is passed to the abstraction. Second, to evaluate $x + x$ the value of x is required. Thus the closure is fetched from the heap and evaluated. Third, the closure is updated with the result so that when the value of x is required again the expression needs not be recomputed.

Measurements by Marlow show that 70% of all closures are used at most once and that it is therefore unnecessary to update them. Usage information also enables a series of program transformations such as more aggressive inlining and let-floating [TWM95,WPJ99,GS99]. It is therefore no surprise that considerable effort has been put into static analyses that can discover if a closure is used at most once [Ses91,LGH$^+$92,Mar93,TWM95,Fax95,BJ96,Mog97,Gus98,WPJ99]. This line of research has produced analyses with increasing accuracy, and benchmarks have shown that for small programs they discover a large portion of closures used at most once. However these analyses are monovariant and do not take the context where a function is called into account. When analysing large programs it is crucial to take the context into account – when Wansbrough and Peyton Jones implemented the recent analysis from [WPJ99] into the Glasgow

Haskell Compiler they discovered that it was almost useless in practice since it did not scale up for large programs. [WPJ00].

In this paper we present a powerful and accurate type system which attempts to solve this problem. It takes the context where a function is called into account through bounded usage polymorphism. We designed our type system by putting together and extending the best ideas from previous work. The salient features of the type system are these:

- Our system has full-blown bounded usage polymorphism and supports usage polymorhic recursion.
- In [WPJ98] Wansbrough and Peyton Jones give an overview of the design space for how to treat data structures. We choose the most aggressive approach which corresponds to the hard-wired treatment of lists in [TWM95].
- Our system is based on subsumption between usage types. The use of subtyping in usage analysis goes back to Faxén [Fax95].
- We have a three-level type language which incorporates separate notions of usage of closures and usage of values which gives increased precision. To separate the usage of closures and values is an idea due to Faxén [Fax95].
- We have expressive update annotations which allow us to express more aggressive optimisations than previous analyses.

Having all these features is not very useful unless there is an efficient inference algorithm for the type system. Here bounded polymorphism presents a problem. See for example Mossin's thesis [Mos97] for an account of the problems with bounded flow polymorphism in type based flow analyses. The core of the problem is that the quantified variables in a type schema may be constrained by a huge number of constraints. In the naive inference algorithm first presented by Mossin the number of constraints may be exponential in the size of the program. Mossin refines the algorithm by adding a constraint simplification phase which renders an inference algorithm which is $O(n^7)$.

A novelty in our work is a new expressive form of constraints which allows constraints to be represented compactly through calls to *constraint abstractions.* To efficiently compute least solutions to constraints with constraint abstractions is an involved problem and is the subject of a companion paper [GS01]. There we show how to efficiently compute a least solution to constraints in a constraint language with constraint abstractions and inequality constraints over a lattice. Using these techniques we can obtain an inference algorithm for our usage analysis which is $O(n^3)$ where n is the size of the explicitly typed program. We believe that constraint abstractions can be very useful for a range of program analyses which features bounded annotation polymorphism and in [GS01] we show how to apply the ideas to a flow analysis with bounded flow polymorphism. Other candidates may be effect analysis, e.g., [TJ94], binding time analysis, e.g., [DHM95], non determinism analysis, e.g., [PS00] and uniqueness type systems, e.g., [BS96].

1.1 Outline

This paper is organised as follows. Section 2 introduces the language and its semantics. Section 3 presents the type system. Section 4 describes related work. Section 5 concludes.

2 Language

In this section we will present our language and its semantics in the form of an abstract machine.

2.1 Syntax

The language we use is a lambda calculus extended with integers, lists, case-expressions and recursive let-expressions. We omit user defined data structures to simplify the presentation but it is a straightforward matter to add them [Sve00].

Variables		x, y, z
Values	v	$::= \lambda x.e \mid n \mid \texttt{nil} \mid \texttt{cons}\ x\ y$
Expressions	e	$::= v^\kappa \mid x \mid e\ x \mid e_0 +^\kappa e_1 \mid \texttt{let}\ b_1, \ldots, b_n\ \texttt{in}\ e \mid \texttt{case}\ e\ \texttt{of}\ alts$
Bindings	b	$::= x =^\kappa e$
Alternatives	$alts$	$::= \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\ x\ y \Rightarrow e_1\}$
Annotations	κ	$::= 1 \mid \omega$

We annotate bindings, values and $+$ with usage annotations 1 and ω ranged over by κ. The intuitive meaning of 1 and ω is that the annotated binding (or value) may be used at most once and any number of times respectively.

A distinguishing feature of the syntax is that arguments (in applications of terms and constructors) are restricted to variables. We will occasionally use unrestricted application $e_0\, e_1$ as syntactic sugar for $\texttt{let}\, x =^\omega e_1\, \texttt{in}\, e_0\, x$ where x is a fresh variable. The purpose of the restricted syntax is to make the creation of closures explicit via a let-expression which greatly simplifies the presentation of the abstract machine as well as the analysis presented in this paper. The syntactic restriction is by now rather standard, see for example [PJPS96,Lau93,Ses97,GS99].

2.2 Semantics

We will take Sestoft's abstract machine [Ses97] as the semantic basis of our work. The machine can be thought of as modelling lower-level abstract machines based on so called update markers, such as the TIM [FW87] and the STG-machine [PJ92]. A correspondence between Sestoft's machine and Launchbury's natural semantics for lazy evaluation [Lau93] has been shown in [Ses97]. For the purpose of the abstract machine we extend the set of terms to include expressions of the form $\mathtt{add}_n^\kappa\ e$, which represents an intermediate step in the computation of $n^{\kappa'} +^\kappa e$. We define a reduction relation $e \mapsto e'$ between terms:

$$(\lambda x.e)^\kappa\ y \mapsto e[x{:=}y] \qquad n^{\kappa'} +^\kappa e \mapsto \mathtt{add}_n^\kappa\ e \qquad \mathtt{add}_{n_0}^\kappa\ n_1^{\kappa'} \mapsto \lceil n_0 + n_1 \rceil^\kappa$$

$$\begin{pmatrix} \mathtt{case}\ \mathtt{nil}^\kappa\ \mathtt{of} \\ \quad \mathtt{nil} \Rightarrow e_0 \\ \quad \mathtt{cons}\ x'\ y' \Rightarrow e_1 \end{pmatrix} \mapsto e_0 \qquad \begin{pmatrix} \mathtt{case}\ (\mathtt{cons}\ x\ y)^\kappa\ \mathtt{of} \\ \quad \mathtt{nil} \Rightarrow e_0 \\ \quad \mathtt{cons}\ x'\ y' \Rightarrow e_1 \end{pmatrix} \mapsto e_1[x'{:=}x, y'{:=}y]$$

Note that no reduction depends on an annotation. The annotations are instead taken into account in the abstract machine transition rules.

Configurations in the abstract machine are triples $\langle H\ ;\ e\ ;\ S\rangle$, where H is a heap, e is the term currently being evaluated and S is the abstract machine stack:

$$\begin{array}{lrl} \text{Heaps} & H & ::= b_1, \ldots, b_n \\ \text{Stacks} & S & ::= \epsilon \mid R, S \mid \#x, S \\ \text{Reduction contexts} & R & ::= [\cdot]\ x \mid [\cdot] +^\kappa e \mid \mathtt{add}_n^\kappa\ [\cdot] \mid \mathtt{case}\ [\cdot]\ \mathtt{of}\ \mathit{alts} \end{array}$$

A heap consists of a sequence of bindings. The variables bound by the heap must be distinct and the order of bindings is irrelevant. Thus a heap can be considered as a partial function mapping variables to terms and we will write $\mathrm{dom}(H)$ for the set of variables bound by H. We will write H_0, H_1 for the concatenation of H_0 and H_1. An abstract machine stack is a stack of shallow reduction contexts and update markers. The stack can be thought of as corresponding to the "surrounding derivation" in a natural semantics, where the rôle of an update marker $\#x$ is to keep track of a pending update of x. The update markers on the stack will be distinct, that is there will be no more than one pending update of the same variable. We will consider an update marker as a binder and we will write $\mathrm{dom}(S)$ for the variables bound by the update markers in S. Consequently, we will require the variables bound by the stack to be distinct from the variables bound by the heap. We will also require that configurations are closed and we will identify configurations up to α-conversion, that is renaming of the variables bound by the heap and the stack. We will also identify configurations up to garbage meaning that we may remove or add bindings and update markers to the heap as long as the configuration remains closed. An initial configuration is of the form $\langle \epsilon\ ;\ e\ ;\ \epsilon\rangle$, where e is a closed expression. The transition rules of the abstract machine are given in Figure 1. The rule Let

$$\langle H\ ;\ \mathtt{let}\ \vec{b}\ \mathtt{in}\ e\ ;\ S\rangle \overset{\text{Let}}{\longmapsto} \langle H, \vec{b}\ ;\ e\ ;\ S\rangle$$

creates new bindings in the heap. For the rule to be applied the variables bound by $\vec{b}$ must be distinct from the variables bound by H and S. This condition can always be met simply by α-converting the let-expression. The rule Var-ω

$$\langle H, x =^\omega e\ ;\ x\ ;\ S\rangle \overset{\text{Var-}\omega}{\longmapsto} \langle H\ ;\ e\ ;\ \#x, S\rangle$$

gives semantics to bindings annotated with ω. The rule states that an update marker shall be pushed onto the stack so that the variable x eventually may be

$$
\begin{array}{lcl}
\langle H\ ;\ \mathtt{let}\,\vec{b}\,\mathtt{in}\,e\ ;\ S\rangle & \overset{\text{Let}}{\longmapsto} & \langle H,\vec{b}\ ;\ e\ ;\ S\rangle \\
\langle H, x =^{\omega} e\ ;\ x\ ;\ S\rangle & \overset{\text{Var-}\omega}{\longmapsto} & \langle H\ ;\ e\ ;\ \#x, S\rangle \\
\langle H, x =^{1} e\ ;\ x\ ;\ S\rangle & \overset{\text{Var-1}}{\longmapsto} & \langle H\ ;\ e\ ;\ S\rangle \\
\langle H\ ;\ R[e]\ ;\ S\rangle & \overset{\text{Unwind}}{\longmapsto} & \langle H\ ;\ e\ ;\ R, S\rangle \\
\langle H\ ;\ v^{\kappa}\ ;\ R, S\rangle & \overset{\text{Reduce}}{\longmapsto} & \langle H\ ;\ e\ ;\ S\rangle \quad \text{if } R[v^{\kappa}] \mapsto e \\
\langle H\ ;\ v^{\omega}\ ;\ \#x, S\rangle & \overset{\text{Marker-}\omega}{\longmapsto} & \langle H, x =^{\omega} v^{\omega}\ ;\ v^{\omega}\ ;\ S\rangle \\
\langle H\ ;\ v^{1}\ ;\ \#x, S\rangle & \overset{\text{Marker-1}}{\longmapsto} & \langle H\ ;\ v^{1}\ ;\ S\rangle
\end{array}
$$

Fig. 1. Abstract machine transition rules

updated with the result of evaluating e. The removal of the binding corresponds to so called black-holing: if the evaluation of e to a value depends on x (i.e., x depends directly on itself) the computation will get stuck, since x is no longer bound by the heap. Note that we still consider the configuration to be closed, since x is bound by the update marker on the stack. The rule Var-1

$$\langle H, x =^{1} e\ ;\ x\ ;\ S\rangle \overset{\text{Var-1}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle$$

gives semantics to bindings annotated with 1. Such bindings may only be used once so there is no need to update the binding and thus no update marker is pushed onto the stack. Note that we require configurations to be closed so the rule does not apply unless the configuration remains closed. An example of where the rule does not apply is the configuration

$$\langle x =^{1} 1 +^{\omega} 2\ ;\ x\ ;\ [\cdot] +^{\kappa} x, \epsilon\rangle$$

which cannot reduce further since there is a reference to x on the stack. This restriction is important since an open configuration would correspond to dangling pointers in an implementation. If the rule does not apply the computation will go *wrong*, and we will consider the configuration and the term it originates from to be ill-annotated. The key property of the type system presented in this paper is that if a term is well-typed then it cannot go wrong. Note that, the insistence that configurations remain closed is a stronger requirement than the intuitive "used at most once" criterion, which says that it is safe to avoid updating a closure if it is used at most once. For example, according to the weaker criterion it is safe to not update x in

$$\mathtt{let}\ x = 1 + 2\ \mathtt{in}\ x + (\lambda y.3)\ x$$

because x is only used once, but according to our criterion it is not safe. Our stronger criterion is useful for two reasons. Firstly, with dangling pointers special care has to be taken so that the garbage collector does not follow them – and there is a cost associated with that. Secondly, usage annotations can be used to justify certain program transformations, such as more aggressive inlining. Gustavsson and Sands [GS99] have shown that the stronger criterion can guarantee that these transformations are time and space safe, but with the weaker "used at most once" criterion the transformations can lead to an asymptoticly worse space behaviour. The rule Unwind

$$\langle H\ ;\ R[e]\ ;\ S\rangle \overset{\text{Unwind}}{\longmapsto} \langle H\ ;\ e\ ;\ R, S\rangle$$

allows us to get to the heart of the evaluation by "unwinding" a shallow reduction context. When the term to be evaluated is a value the next transition depends on whether an update marker or a reduction context is on top of the stack. If it is a reduction context the rule Reduce

$$\langle H\ ;\ v\ ;\ R, S\rangle \overset{\text{Reduce}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle \quad \text{if } R[v] \mapsto e$$

applies, the value is plugged into the reduction context and a reduction can take place. If the top of the stack is an update marker, what happens depends on the annotation on the value. If it is ω the value may be used several times and we apply the rule Update-ω

$$\langle H\ ;\ v^{\omega}\ ;\ \#x, S\rangle \overset{\text{Marker-}\omega}{\longmapsto} \langle H, x =^{\omega} v^{\omega}\ ;\ v^{\omega}\ ;\ S\rangle$$

which takes care of the update marker and performs the update. If the value on the other hand is annotated with 1, the value may only be used once so the rule Update-1

$$\langle H\ ;\ v^{1}\ ;\ \#x, S\rangle \overset{\text{Marker-1}}{\longmapsto} \langle H\ ;\ v^{1}\ ;\ S\rangle$$

throws away the marker without performing the update. Again, note that the rule does not apply unless the configuration remains closed. So, for example,

$$\langle \epsilon\ ;\ 3^{1}\ ;\ \#x, [\cdot] +^{\kappa} x, \epsilon\rangle$$

goes wrong and we consider the configuration to be ill-annotated.

3 Type system

The semantics in Section 2 specifies that for a binding $x{=}e$ to be safely annotated with a 1 it is required that whenever the binding is used through the rule

$$\langle H, x =^{1} e\ ;\ x\ ;\ S\rangle \overset{\text{Var-1}}{\longmapsto} \langle H\ ;\ e\ ;\ S\rangle,$$

the configuration must remain closed. Thus there may only be one (non-binding) occurrence of x in the configuration, namely the one that is dereferenced. Similarly, to safely annotate a value with 1 it is required that if and when the value is used and there is an update marker $\#x$ on the stack

$$\langle H\ ;\ v^{1}\ ;\ \#x, S\rangle \overset{\text{Marker-1}}{\longmapsto} \langle H\ ;\ v^{1}\ ;\ S\rangle \quad ,$$

then there is no live occurrence of x in the configuration so that the configuration remains closed. Our type system (and most other type based usage analyses) is based on the following simple idea. If, when a binding $x = e$ is created, x occurs only once in the configuration and x never gets duplicated during the computation then x will occur only once if and when it is dereferenced. [1]

3.1 Type language

In order to construct a type system for the annotated language we need a corresponding annotated type language. We start by extending the annotation language from the previous section to include annotation variables.

$$\text{Annotations } \kappa ::= 1 \mid \omega \mid k \mid j$$

We will use two kinds of variables, *type annotation variables*, ranged over by k, and *program annotation variables*, ranged over by j. Type annotation variables may occur in the annotations on a type but not in the annotations on a program. Conversely, program annotation variables may occur in programs but not in types.

The structure of the type language closely follows the structure of the term language and we will have one kind of type for every syntactic category. We let ρ range over *value types* which is the form of type we will assign to values.

$$\begin{array}{ll} \text{Type Variables } a & \\ \text{Value Types} & \rho ::= a \mid \texttt{Int} \mid \sigma \to \tau \mid \texttt{List}\ \kappa_0\ \kappa_1\ \kappa_2\ \kappa_3\ \rho \end{array}$$

Our value types contains type variables, an integer type, function types and the list type. The function types relies on a notion of *binding types*, ranged over by σ, and *expression types*, ranged over by τ, which we will introduce below. Expression types are used to give types to expressions and are defined as follows.

$$\text{Expression Types } \tau ::= \rho^{\kappa}$$

An annotated value v^{κ} will be given a type of the form ρ^{κ} and a non-value e will be given a type such that the annotated value of e (if e terminates) will have that type. Thus, for example, saying that a term has a type ρ^{ω} means that the value of the term may be used any number of times. Binding types which we will use to give a type to bindings are defined as follows.

$$\text{Binding Types } \sigma ::= \tau_{\kappa}$$

A binding $x =^{\kappa} e$ may be given a type of the form τ_{κ} where τ is the type of e. We also use binding types to give a type to a variable when we can think of the variable as a reference, for example when we pass it as an argument to a

[1] We will strengthen this idea in an obvious but important way – when a variable occurs once in several branches of a case-expression. Then, since eventually only one branch will be taken, we may consider it as occurring only once.

function. A type of a variable is then simply the type of the bindings it may refer to. Recall that we used expression types and binding types in the type $\sigma \to \tau$ of a function. A function of this type can be applied to a variable (remember functions can only be applied to variables due to the syntactic restriction in our language) with the binding type σ and then it will return something of type τ. We can also use binding types to logically justify our type $\texttt{List}\,\kappa_0\,\kappa_1\,\kappa_2\,\kappa_3\,\rho$ of lists. We can obtain this type simply by annotating the right hand side of the data type definition

$$\texttt{List}\ a = \texttt{nil} \mid \texttt{cons}\ a\ (\texttt{List}\ a)$$

such that the arguments to the constructors are binding types, as follows.

$$\texttt{List}\ k_0\ k_1\ k_2\ k_3\ a = \texttt{nil} \mid \texttt{cons}\ a_{k_0}^{k_1}\ (\texttt{List}\ k_0\ k_1\ k_2\ k_3\ a)_{k_2}^{k_3}$$

The reason for why the arguments to the constructors should be binding types is simply because constructors, due to the syntactic restriction, may be applied only to variables.

3.2 Subtyping

A key observation which we will use to justify our subtyping relation is that 1 operationally approximates ω, i.e., if we in any term e replace any occurrence of 1 with ω then the modified term will run successfully without going wrong if and when e does. We define the subtyping relation on *closed* types where the ordering on annotations is the operational approximation $1 < \omega$ by the following rules.

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'} \qquad \frac{\rho_{\kappa_0}^{\kappa_1} \leq {\rho'}_{{\kappa_0}'}^{{\kappa_1}'} \qquad \kappa_2' \leq \kappa_2 \qquad \kappa_3' \leq \kappa_3}{\texttt{List}\ \kappa_0\ \kappa_1\ \kappa_2\ \kappa_3\ \rho \leq \texttt{List}\ {\kappa_0}'\ {\kappa_1}'\ {\kappa_2}'\ {\kappa_3}'\ \rho'}$$

$$\frac{}{\texttt{Int} \leq \texttt{Int}} \qquad \frac{\rho \leq \rho' \qquad \kappa' \leq \kappa}{\rho^{\kappa} \leq {\rho'}^{\kappa'}} \qquad \frac{\tau \leq \tau' \qquad \kappa' \leq \kappa}{\tau_{\kappa} \leq {\tau'}_{\kappa'}}$$

Note that the subtype ordering is contravariant with respect to the ordering on the annotations. The rule for lists can be understood by unfolding the annotated data type definition for lists.

3.3 Constraints

In order to extend the subtyping relation to types with type variables and annotation variables we need the notion of constraints. To be able to represents constraints compactly we introduce a new form of constraints which may contain calls to *constraint abstractions.* A constraint abstraction is simply a function that given some annotation variables returns a constraint. We will let ϕ range over constraint abstractions, l range over constraint abstraction variables and Π range over constraints.

$$\begin{array}{lrl} \text{Annotation constraints} & \Pi & ::= \kappa_0 \leq \kappa_1 \mid \Pi_0, \Pi_1 \mid \text{let}\ \vec{\phi}\ \text{in}\ \Pi \mid \exists \vec{k}.\Pi \mid l\ \vec{\kappa} \\ \text{Constraint abstractions} & \phi & ::= l\ \vec{k} = \Pi \end{array}$$

Constraint abstractions allow different substitution instances of a constraint to share the same representation. For example to represent instances of the constraints $k_0 \leq k_1,\ k_1 \leq k_2$ we can define an abstraction

$$l\ k_0\ k_1\ k_2 = k_0 \leq k_1,\ k_1 \leq k_2$$

and represent $(\kappa_0 \leq \kappa_1,\ \kappa_1 \leq \kappa_2), (\kappa_3 \leq \kappa_4,\ \kappa_4 \leq \kappa_5)$ as

$$\text{let}\ l\ k_0\ k_1\ k_2 = k_0 \leq k_1,\ k_1 \leq k_2\ \text{in}\ l\ \kappa_0\ \kappa_1\ \kappa_2,\ l\ \kappa_3\ \kappa_4\ \kappa_5.$$

Thus with constraint abstractions the size of any instance is linear in the number of free type annotation variables of the constraint but the size of the original constraint may be quadratic in the sum of the number of free type annotation variables and free program annotation variables (or even worse if it contains existential quantifiers). With constraint abstraction we can avoid the exponential explosion of constraints which can happen with a naive approach. To see why consider a program of the following form.

$$\begin{array}{l}
\texttt{let}\ f_0 = \ \ldots \\
\texttt{in}\ \texttt{let}\ f_1 = \ \ldots f_0 \ldots f_0 \ldots \\
\quad \texttt{in}\ \texttt{let}\ \ldots \\
\qquad \texttt{in}\ \texttt{let}\ f_n = \ \ldots f_{n-1} \ldots f_{n-1} \ldots \\
\qquad\quad \texttt{in}\ \ldots f_n \ldots f_n \ldots
\end{array}$$

The first naive algorithm, for the similar problem of flow analysis with bounded flow polymorphism, presented by Mossin [Mos97] which suffers from the exponential explosion problem would proceed as follows. It first infers the polymorphic type for f_0. Then to compute the type for f_1 it instantiates the type of f_0 twice and thus make two instances of the constraints contained in the type schema so the constraints for f_1 will be at least twice as big. This is repeated n times and thus the size of the resulting constraints will be exponential in the call depth n. In practice the call depth typically does not grow linearly with the size of the program but the call depth does tend to increase with program size which makes this into a problem that occurs in practice. With constraint abstractions we can avoid the problem and represent the constraints as follows

$$\begin{array}{l}
\text{let}\ l_0\ \vec{k}_0 = \ \ldots \\
\text{in let}\ l_1\ \vec{k}_1 = \ \ldots l_0\ \vec{k}'_0 \ldots l_0\ \vec{k}''_0 \ldots \\
\quad \text{in let}\ \ldots \\
\qquad \text{in let}\ l_n\ \vec{k}_n = \ \ldots l_{n-1}\ \vec{k}'_{n-1} \ldots l_{n-1}\ \vec{k}''_{n-1} \ldots \\
\qquad\quad \text{in}\ \ldots l_n\ \vec{k}'_0 \ldots l_n\ \vec{k}''_0 \ldots
\end{array}$$

To give semantics to constraints we will use closing substitutions from type variables to value types and annotation variables to annotations, ranged over by ϑ. The meaning of a constraint Π is given by a relation $\vartheta; \vec{\phi} \models \Pi$ (read as $\vartheta; \vec{\phi}$

models Π) defined coinductively by the following rules.

$$\frac{\kappa_0\vartheta \le \kappa_1\vartheta}{\vartheta; \vec{\phi} \models \kappa_0 \le \kappa_1} \qquad \frac{\vartheta; \vec{\phi} \models \Pi_0 \qquad \vartheta; \vec{\phi} \models \Pi_1}{\vartheta; \vec{\phi} \models \Pi_0, \Pi_1} \qquad \frac{\vartheta; \vec{\phi}, \vec{\phi}' \models \Pi}{\vartheta; \vec{\phi} \models \text{let } \vec{\phi}' \text{ in } \Pi}$$

$$\frac{\vartheta; \vec{\phi} \models \Pi[\vec{k} := \vec{\kappa}]}{\vartheta; \vec{\phi} \models \exists \vec{k}.\Pi} \qquad \frac{\vartheta; \vec{\phi} \models \Pi[\vec{k} := \vec{\kappa}]}{\vartheta; \vec{\phi} \models l\,\vec{\kappa}} \quad l\,\vec{k} = \Pi \in \vec{\phi}$$

We will sometimes write $\vartheta \models \Pi$ as a shorthand for $\vartheta; \epsilon \models \Pi$. We will let Ψ range over constraints concerning type variables.

$$\text{Type variable constraints } \Psi \ ::= a_0 \le a_1 \mid \Psi_0, \Psi_1 \mid \exists \vec{a}.\Psi$$

The meaning of a constraint Ψ is given by a relation $\vartheta \models \Psi$ (read as ϑ models Ψ). We define $\vartheta \models \Psi$ inductively by the following rules.

$$\frac{\vartheta(a_0) \le \vartheta(a_1)}{\vartheta \models a_0 \le a_1} \qquad \frac{\vartheta \models \Psi_0 \qquad \vartheta \models \Psi_1}{\vartheta \models \Psi_0, \Psi_1} \qquad \frac{\vartheta[\vec{a} := \vec{\rho}] \models \Psi}{\vartheta \models \exists \vec{a}.\Psi}$$

We will let Θ range over pairs $\Pi; \Psi$ and we define $\vartheta \models \Theta$ as $\vartheta \models \Pi; \Psi$ iff $\vartheta \models \Pi$ and $\vartheta \models \Psi$. The whole purpose of having constraints is that they allow us to extend the subtyping relation to types with variables. We will define a relation $\Theta \models \rho_0 \le \rho_1$ where ρ_0 and ρ_1 may be open types, which reads: $\rho_0 \le \rho_1$ is a consequence of Θ. It is defined as $\Theta \models \rho_0 \le \rho_1$ iff for every ϑ, if $\vartheta \models \Theta$ then $\rho_0\vartheta \le \rho_1\vartheta$. We also define $\Theta \models \tau_0 \le \tau_1$ and $\Theta \models \sigma_0 \le \sigma_1$ in the same manner.

3.4 Type schemas

Our type system incorporates bounded polymorphism so we need type schemas where the quantified variables are bounded by some constraints.

$$\text{Type Schemas } \chi \ ::= \forall \vec{k}, \vec{a}.\, \rho \mid \Theta$$

We will define a relation $\Theta \models \chi \prec \rho$ which reads as: it is a consequence of Θ that χ can be instantiated to ρ. It is defined as $\Theta \models (\forall \vec{k}, \vec{a}.\, \rho \mid \Theta') \prec \rho[\vec{k} := \vec{\kappa}, \vec{a} := \vec{\rho}]$ iff for every ϑ, if $\vartheta \models \Theta$ then $\vartheta \circ [\vec{k} := \vec{\kappa}, \vec{a} := \vec{\rho}] \models \Theta'$. We will sometimes consider a value type ρ to be a type schema with no quantified variables and no constraints.

3.5 Contexts

We use Γ and Δ to range over typing contexts which are multisets of type associations of the form $x : \chi_{\kappa}^{\kappa'}$ (and since we may consider a value type ρ as a type schema there may also be type associations of the form $x : \rho_{\kappa}^{\kappa'}$). As usual we will use contexts when we give a type to a term with free variables. Thus we will say that e has the type τ in a context Γ if we can give e the type τ

assuming that the free variables in e has the types given by Γ. However the context also plays another important rôle; it records the number of times each variable occurs in the term. Thus if x occurs n times in e it also occurs n times in Γ (with one important exception, namely if x occurs in different branches of a case-expression). This may be a bit surprising at first. Consider for example the term $(\lambda y.y +^1 y)^1\ x$ with the free variable x. We will be able to say that this term has the type $\mathtt{Int}^1$ in the context $x : \mathtt{Int}^\omega_\omega$. According to the reduction relation the term can reduce to $x +^1 x$ so we would expect to be able to give $x+^1 x$ the same type in the same context. However this will not be possible since x now occurs twice in the term. Instead we can type the term in the context $x : \mathtt{Int}^\omega_\omega, x : \mathtt{Int}^\omega_\omega$ where x occurs twice. To be able to state a relation between the contexts before and after a reduction we define a rewrite relation on contexts.

$$\Gamma,\ x : \chi^\omega_\omega \to \Gamma,\ x : \chi^\omega_\omega,\ x : \chi^\omega_\omega \qquad \Gamma,\ x : \chi^{\kappa'}_\kappa \to \Gamma$$

We have two rewrite rules. The first says that a type association of the form $x : \chi^\omega_\omega$ may be duplicated. This is supposed to model the duplication of a variable x during the computation. Note that we may not duplicate a type association of the form $x : \chi^1_1$. This reflects our intention that a variable that refers to a binding which will not be updated, must not be duplicated. The second rule simply allows us to remove a type association. This corresponds to the case when a variable is dropped during the computation (for example since it occurred in a branch of a case-expression that was not selected). These rewrite rules will play a rôle similar to the contraction and weakening rules in logic. The restricted duplication (i.e., that we may only duplicate type associations of the form $x : \chi^\omega_\omega$) corresponds to the restricted form of contraction in linear logic [Gir87]. We extend the relation to contexts with open types in the same way as with the subtyping relation by defining $\Theta \models \Gamma_0 \to^* \Gamma_1$ iff for every ϑ, if $\vartheta \models \Theta$ then $\Gamma_0\vartheta \to^* \Gamma_1\vartheta$. Finally we will also need the relation $\Theta \models \mathtt{if}\ \kappa = \omega\ \mathtt{then}\ \Gamma \to^* \Gamma, \Gamma$ which holds iff for every ϑ, if $\vartheta \models \Theta$, and $\kappa\vartheta = \omega$ then $\Gamma\vartheta \to^* \Gamma\vartheta, \Gamma\vartheta$.

3.6 Typing judgements

Typing judgements for values take the form $\Theta; \Gamma \vdash v : \rho$ and shall be read: under the constraints Θ and in the context Γ, the value v can be given the value type ρ. Similarly we will have typing judgements for expressions, alternatives and bindings. As discussed in the previous section the context Γ in our judgements as usual keeps track of the types of the free variables in the term but it also records the number of times each variable occurs in the term.

3.7 Typing rules

The typing rules for values are in Figure 2. The key feature of the rule Abs

$$\frac{\Theta; \Gamma_0, \Gamma_1 \vdash e : \tau}{\Theta; \Gamma_0 \vdash \lambda x.e : \sigma \to \tau} \quad \begin{array}{l} x \notin \mathrm{dom}(\Gamma_0) \\ \Theta \models x : \sigma \to^* \Gamma_1 \end{array}$$

$$\text{Abs}\frac{\Theta;\Gamma_0,\Gamma_1 \vdash e:\tau}{\Theta;\Gamma_0 \vdash \lambda x.e:\sigma \to \tau}\quad \begin{array}{l} x \notin \mathrm{dom}(\Gamma_0) \\ \Theta \models x:\sigma \to^* \Gamma_1 \end{array}$$

$$\text{Int}\frac{}{\Theta;\emptyset \vdash n:\mathtt{Int}} \qquad \text{Nil}\frac{}{\Theta;\emptyset \vdash \mathtt{nil}:\mathtt{List}\ \kappa_0\ \kappa_1\ \kappa_2\ \kappa_3\ \rho}$$

$$\text{Cons}\frac{}{\Theta; x:\chi_0{}^{\kappa_1}_{\kappa_0}, y:\chi_1{}^{\kappa_3}_{\kappa_2} \vdash \mathtt{cons}\ x\ y:\rho'}\quad \begin{array}{l} \rho' \equiv \mathtt{List}\ \kappa_4\ \kappa_5\ \kappa_6\ \kappa_7\ \rho \\ \Theta \models \chi_0 \prec \rho_0,\ \chi_1 \prec \rho_1 \\ \Theta \models \rho_0{}^{\kappa_1}_{\kappa_0} \le \rho^{\kappa_5}_{\kappa_4},\ \rho_1{}^{\kappa_3}_{\kappa_2} \le \rho'^{\kappa_7}_{\kappa_6} \end{array}$$

Fig. 2. Typing rules for values

is that if x occurs more than once in e then the abstraction will be assigned a type of the form $\rho^{\kappa}_{\kappa'} \to \tau$ where κ and κ' are constrained to be ω indicating that a variable will be duplicated if it is passed to the abstraction. This is accomplished by first typing e in a context Γ_0, Γ_1 where $x \notin \mathrm{dom}(\Gamma_0)$. Then, if x occurs more than once in e, x will occur more than once in Γ_1. Now the second side condition specify that we must be able to rewrite $x : \rho^{\kappa}_{\kappa'}$ to Γ_1 which clearly involves duplicating $x : \rho^{\kappa}_{\kappa'}$ (since x occurs more than once in Γ_1) which will constrain κ and κ' to be ω. The typing rule for integers is straightforward and the rules for lists can be understood by unfolding the annotated data type definition for lists.

We have divided the typing rules for expressions into two figures. Most rules appear in Figure 3 but the rules which concern let expressions are in Figure 4. The rule Value

$$\frac{\Theta;\Gamma \vdash v:\rho}{\Theta;\Gamma \vdash v^{\kappa}:\rho^{\kappa'}}\quad \begin{array}{l} \Theta \models \mathtt{if}\ \kappa' = \omega\ \mathtt{then}\ \Gamma \to^* \Gamma,\Gamma \\ \Theta \models \kappa' \le \kappa \end{array}$$

is used to type an annotated value. Saying that an annotated value has the type $\rho^{\kappa'}$ means that if κ' is ω the value may be used any number of times and thus it will take care of any update marker on the stack. Taking care of an update marker means updating with the value, thus duplicating any free variables of the value. The purpose of the side condition $\Theta \models \mathtt{if}\ \kappa' = \omega\ \mathtt{then}\ \Gamma \to^* \Gamma, \Gamma$ is to ensure that these variables may safely be duplicated if κ' is constrained to be ω.

In order to type case-expressions we introduce an auxiliary form of judgements for alternatives. We give alternatives a type of the form $\rho \Rightarrow \tau$ where ρ is the type of the value that is being scrutinised and τ is the type of the branches. The rule Alts

$$\frac{\Theta;\Gamma_0,\Gamma_1 \vdash e_0:\tau \qquad \Theta;\Gamma_0,\Gamma_2,\Gamma_3 \vdash e_1:\tau}{\Theta;\Gamma_0,\Gamma_1,\Gamma_2 \vdash \{\mathtt{nil} \Rightarrow e_0; \mathtt{cons}\ x\ y \Rightarrow e_1\}:\rho' \Rightarrow \tau}\quad \begin{array}{l} \rho' \equiv \mathtt{List}\ \kappa_0\ \kappa_1\ \kappa_2\ \kappa_3\ \rho \\ x,y \notin \mathrm{dom}(\Gamma_0,\Gamma_2) \\ \Theta \models x:\rho^{\kappa_1}_{\kappa_0}, y:\rho'^{\kappa_3}_{\kappa_2} \to^* \Gamma_3 \end{array}$$

for alternatives contains a subtle treatment of contexts. If a variable occurs once in each branch of the case-expression and thus twice in the term it may still occur

$$\text{Value}\frac{\Theta; \Gamma \vdash v : \rho}{\Theta; \Gamma \vdash v^{\kappa} : \rho^{\kappa'}} \quad \begin{array}{l}\Theta \models \texttt{if } \kappa' = \omega \texttt{ then } \Gamma \to^{*} \Gamma, \Gamma \\ \Theta \models \kappa' \leq \kappa\end{array}$$

$$\text{Var}\frac{}{\Theta; x : \chi^{\kappa_1}_{\kappa_0} \vdash x : \tau} \quad \begin{array}{l}\Theta \models \chi \prec \rho \\ \Theta \models \rho^{\kappa_1} \leq \tau\end{array} \qquad \text{App}\frac{\Theta; \Gamma \vdash e : (\rho^{\kappa_0}_{\kappa_1} \to \tau)^{\kappa}}{\Theta; \Gamma, x : \chi^{\kappa_0}_{\kappa_1} \vdash e\, x : \tau} \quad \Theta \models \chi \prec \rho$$

$$\text{Plus}\frac{\Theta; \Gamma_0 \vdash e_0 : \texttt{Int}^{\kappa_0} \qquad \Theta; \Gamma_1 \vdash e_1 : \texttt{Int}^{\kappa_1}}{\Theta; \Gamma_0, \Gamma_1 \vdash e_0 +^{\kappa} e_1 : \texttt{Int}^{\kappa'}} \quad \Theta \models \kappa' \leq \kappa$$

$$\text{Alts}\frac{\Theta; \Gamma_0, \Gamma_1 \vdash e_0 : \tau \qquad \Theta; \Gamma_0, \Gamma_2, \Gamma_3 \vdash e_1 : \tau}{\Theta; \Gamma_0, \Gamma_1, \Gamma_2 \vdash \{\texttt{nil} \Rightarrow e_0; \texttt{cons}\; x\; y \Rightarrow e_1\} : \rho' \Rightarrow \tau} \quad \begin{array}{l}\rho' \equiv \texttt{List}\; \kappa_0\; \kappa_1\; \kappa_2\; \kappa_3\; \rho \\ x, y \notin \mathrm{dom}(\Gamma_0, \Gamma_2) \\ \Theta \models x : \rho^{\kappa_1}_{\kappa_0}, y : \rho'^{\kappa_3}_{\kappa_2} \to^{*} \Gamma_3\end{array}$$

$$\text{Case}\frac{\Theta; \Gamma_0 \vdash e : \rho^{\kappa} \qquad \Theta; \Gamma_1 \vdash \mathit{alts} : \rho \Rightarrow \tau}{\Theta; \Gamma_0, \Gamma_1 \vdash \texttt{case}\; e\; \texttt{of}\; \mathit{alts} : \tau}$$

Fig. 3. Typing rules for expressions

only once in the context. This is achieved by collecting the variables that occur in both branches in a common context Γ_0, thus effectively counting a variable occurring in both branches as one. Finally, the side conditions take care of the variables bound in the cons-pattern. They see to that if x (and/or y) occurs several times in e_1 then κ_0 and κ_1 (and/or κ_2 and κ_3) will be constrained to be ω. Thanks to the auxiliary rule for alternatives the rule for case-expressions becomes entirely straightforward.

To type let-expressions we first introduce an auxiliary form of typing judgements for bindings. We will give bindings a type of the form $x : \chi^{\kappa_0}_{\kappa_1}$, i.e., the type of a binding includes the name of the bound variable (so it can be considered as a type association). The rules for typing bindings appears in Figure 4. To type a binding with the rule Binding

$$\frac{\Pi_0; \Psi; \Gamma \vdash e : \rho^{\kappa_0}}{\Pi; \Gamma \vdash x =^{\kappa} e : (x : (\forall \vec{k}_1, \vec{a}_1 .\, \rho \,|\, l\vec{k}_2; \exists \vec{a}_0 . \Psi)^{\kappa_0}_{\kappa_1}) \textbf{ where } l\vec{k}_2 = \exists \vec{k}_0 . \Pi_0} \quad (*)$$

$$(*)\; \begin{array}{l}\vec{k}_0 \notin \mathit{ftav}(\Gamma, \rho^{\kappa}_0),\ \vec{a}_0 \notin \mathit{ftv}(\Gamma, \rho^{\kappa}_0), \\ \vec{k}_1 \notin \mathit{ftav}(\Gamma, \kappa_0,\ l\vec{k}_2 = \exists \vec{k}_0 . \Pi_0), \\ \vec{a}_1 \notin \mathit{ftv}(\Gamma),\ \Pi \models \kappa_1 \leq \kappa\end{array}$$

we first type the expression in the binding and yield the constraints $\Pi_0; \Psi$. We may then existentially quantify variables which appear in the constraints to obtain $\exists \vec{k}_0 . \Pi_0$ and $\exists \vec{a}_0 . \Psi$ providing $\vec{k}_0$ and $\vec{a}_0$ do not occur free elsewhere in the judgement. This is ensured by the first line of side conditions. We then form the type schema $\forall \vec{k}_1, \vec{a}_1 .\, \rho \,|\, l\vec{k}_2; \exists \vec{a}_0 . \Psi$ by universally quantifying $\vec{k}_1$ and $\vec{a}_1$. The second line of side conditions simply ensures that $\vec{k}_1$ and $\vec{a}_1$ do not occur free

$$\text{Binding}\;\frac{\Pi_0; \Psi; \Gamma \vdash e : \rho^{\kappa_0}}{\Pi; \Gamma \vdash x =^{\kappa} e : (x : (\forall \vec{k}_1, \vec{a}_1.\, \rho \mid l\vec{k}_2; \exists \vec{a}_0.\Psi)^{\kappa_0}_{\kappa_1})\ \mathbf{where}\ l\vec{k}_2 = \exists \vec{k}_0.\Pi_0}\quad (*)$$

$$\text{Binding group-}\epsilon\;\frac{}{\Pi; \epsilon \vdash \epsilon : \epsilon\ \mathbf{where}\ \epsilon} \qquad (*)\ \begin{array}{l} \vec{k}_0 \notin ftav(\Gamma, \rho_0^{\kappa}),\ \vec{a}_0 \notin ftv(\Gamma, \rho_0^{\kappa}), \\ \vec{k}_1 \notin ftav(\Gamma, \kappa_0,\ l\vec{k}_2 = \exists \vec{k}_0.\Pi_0), \\ \vec{a}_1 \notin ftv(\Gamma),\ \Pi \models \kappa_1 \leq \kappa \end{array}$$

$$\text{Binding group}\;\frac{\Pi; \Gamma_0 \vdash b : (x : \chi^{\kappa_1}_{\kappa_0})\ \mathbf{where}\ \phi \qquad \Pi; \Gamma_1 \vdash \vec{b} : \Delta\ \mathbf{where}\ \vec{\phi}}{\Pi; \Gamma_0, \Gamma_1 \vdash b,\ \vec{b} : (x : \chi^{\kappa_1}_{\kappa_0}, \Delta)\ \mathbf{where}\ \phi, \vec{\phi}}$$

$$\text{Let}\;\frac{\Pi_0; \Gamma_0, \Gamma_1 \vdash \vec{b} : \Delta\ \mathbf{where}\ \vec{\phi} \qquad \Pi_1; \Psi; \Gamma_2, \Gamma_3 \vdash e : \tau}{\Pi; \Psi; \Gamma_1, \Gamma_3 \vdash \mathtt{let}\, \vec{b}\, \mathtt{in}\, e : \tau} \quad \begin{array}{l} \mathrm{dom}(\Gamma_1, \Gamma_3) \cap \mathrm{dom}(\Delta) = \emptyset \\ \Pi \models \Delta \rightarrow^* \Gamma_0; \Gamma_2 \\ \Pi \models \Pi_0, \mathrm{let}\, \vec{\phi}\, \mathrm{in}\, \Pi_1 \end{array}$$

Fig. 4. Typing rules for bindings and let expressions

elsewhere in the judgement. We put $\exists \vec{a}_0.\Psi$ in the type schema but not $\exists \vec{k}_0.\Pi_0$. Instead we introduce a constraint abstraction $l\vec{k}_2 = \exists \vec{k}_0.\Pi_0$ and put a call to the constraint abstraction into the type schema. We also need a form of judgements for groups of bindings. As you would expect the type of a group of bindings is just a set of type associations (i.e., a typing context) and the typing rules just collect the type associations and the corresponding constraint abstractions. In the rule Let

$$\frac{\Pi_0; \Gamma_0, \Gamma_1 \vdash \vec{b} : \Delta\ \mathbf{where}\ \vec{\phi} \quad \Pi_1; \Psi; \Gamma_2, \Gamma_3 \vdash e : \tau}{\Pi; \Psi; \Gamma_1, \Gamma_3 \vdash \mathtt{let}\, \vec{b}\, \mathtt{in}\, e : \tau} \quad \begin{array}{l} \mathrm{dom}(\Gamma_1, \Gamma_3) \cap \mathrm{dom}(\Delta) = \emptyset \\ \Pi \models \Delta \rightarrow^* \Gamma_0; \Gamma_2 \\ \Pi \models \Pi_0, \mathrm{let}\, \vec{\phi}\, \mathrm{in}\, \Pi_1 \end{array}$$

we first type the bindings which gives a context Δ which contains the type schemas associated with each binding. The first two side conditions ensures that the type schema ${\chi_i}^{\kappa_i'}_{\kappa_i}$ associated with each variable x_i in Δ is consistent with the type of each use of x_i. They also ensures that if x_i may be used more than once then κ_i and κ_i' must be constrained to ω. It is achieved as follows. If x_i occurs more than once in e and the right hand sides of $\vec{b}$ then x_i will also occur more than once in Γ_0, Γ_2. Thus the second side condition will ensure that κ_i and κ_i' is constrained to be ω. The typing of the bindings also gives a group of constraint abstraction $\vec{\phi}$. With the constraint abstraction we form the constraint $\mathrm{let}\, \vec{\phi}\, \mathrm{in}\, \Pi_1$ which by the third side condition must be a consequence of the constraints in the conclusion of the rule.

3.8 Soundness

The soundness of our type system simply says that a well typed program is well annotated, i.e., when we run it in the abstract machine it does not go wrong.

Theorem 1. *If $\Theta;\emptyset \vdash e : \tau$ and $\vartheta \models \Theta$ then $e\vartheta$ cannot go wrong.*

The result is established by extending the type system to abstract machine configurations and then proving a subject reduction result which says that typings are preserved by transitions in the abstract machine. A very similar proof for the type system in [Gus98] is presented in full detail in [Gus99].

3.9 Inference Algorithm

As stated the type system is undecidable since it employs type polymorphic recursion. Our inference algorithm will therefore take a term which is explicitly typed in the underlying ordinary type system and can handle type polymorphic recursion if presented to it through the type annotations. It will first compute a usage typing judgement which is principal with respect to the given typing judgement, i.e., every other usage typing judgement is an instance of the computed judgement if "stripping the annotations" from it yields the judgement in the underlying type system. The second phase of the algorithm then computes the best solution to the constraints in the principal judgement using the techniques described in a companion paper [GS01].

The time complexity of the algorithm is dominated by the cost of the constraint solving in the second phase. We can argue, as follows, that the time complexity of the second phase is $O(n^3)$ where n is the size of the explicitly typed term. Let the *skeleton* of the constraints be the constraints where all occurrences of inequality constraints of the form $\kappa_0 \leq \kappa_1$ have been removed. What remains are the binding occurrences of variables and all calls to constraint abstractions. By inspecting the typing rules we can see that the size of the skeleton of the constraints required to type a program is proportional to the size of the explicitly typed program. Moreover the number of free annotation variables in the constraints are proportional to the size of the program. From these facts and theorem 2 of [GS01] we can conclude that the complexity is $O(n^3)$ where n is the size of the typed program.

For a version of the analysis in this paper without usage-polymorphic recursion we have developed an algorithm based on *non-recursive* constraint abstractions with a worst case complexity of $O(n * m * t^2)$ where n is the size of the untyped lambda lifted version of the program, m is the size of the type of the largest set of (properly) mutually recursive definitions and t is the size of the largest instantiated type [Sve00]. Since m and t typically grow slowly or not at all with program size we expect that algorithm to scale up well in practice.

4 Related Work

There is a rich literature on analyses which aims at avoiding updates. See [Gus99] for a thorough overview. This work especially lends ideas from the type based approach by Turner, Wadler and Mossin [TWM95], and its followups by Gustavsson [Gus98] and Wansbrough and Peyton Jones [WPJ99]. Bounded polymorphism was proposed by Turner, Wadler and Mossin [TWM95] and the idea to use subtyping in usage analysis originates from the work by Faxén [Fax95] (the subtyping in his flow analysis and the directed edges in the post processing achieves the same effect as the subtyping in this paper) although it was independently proposed by Gustavsson [Gus98] and Wansbrough and Peyton Jones [WPJ99].

The analysis which seems to be closest in expressive power to ours is an analysis by Faxén based on an undecidable type based flow analysis [Fax97]. Due to the undecidable nature of the analysis his inference algorithm is not complete with respect to the type system. The algorithm is parametrised by a notion of finite name supply and the larger name-supply the better the algorithm approximates the type system. The exact relationship between the different degrees of approximations computed by his algorithm and our type system is not clear to us.

The aim of this work is to make usage analysis scale up for large programs and in that respect it is most closely related to recent work by Wansbrough and Peyton Jones [WPJ00]. They have also observed that usage polymorphism is crucial for the accuracy of the analysis of large programs but they side-step the difficulties associated with bounded polymorphism. Instead they have a simple usage polymorphism where the quantified variables may not be constrained. This is achieved by an algorithm which eliminates inequality constraints prior to quantification by unifying constrained variables. The drawback of their approach is that as they refrain from using bounded polymorphism, they get an analysis which is rather inaccurate when it comes to data structures. Consider for example the following program fragment.

$$\ldots \text{map square (fromto 1 100)} \ldots$$

The spine of the list produced by *fromto* is consumed linearly by *map* but a type system with their simple usage polymorphism cannot discover it. The reason being that in a system with simple usage polymorphism the usage of the spine must be unified with the usage of the elements and in this case the elements are used more than once. In our system with bounded polymorphism the usage of the spine and the elements need only to constrain each other through an inequality constraint so we can deduce that the spine is used linearly although the elements are not. We believe that this situation is common enough in practice to have a significant effect on the accuracy of the analysis.

That the number of constraints explodes is a problem also for other type based program analyses with bounded polymorphism. In that respect our work is most closely related to the work by Faxén [Fax95], Mossin [Mos97] and Rehof and Fänhdrich [RF01]. Faxén and Mossin present inference algorithms for type based flow analyses which simplifies constraint sets to smaller but equivalent constraint sets. In their recent work on type based flow analysis Rehof and Fänhdrich uses *instantiation constraints* to represent constraints compactly and thus instantiation constraints plays a rôle similar to our constraint abstractions.

5 Conclusions and Future Work

We have presented a powerful and accurate type system for usage analysis with bounded usage polymorphism and subtyping. A key contribution is a new expressive form of constraints which allows constraints to be represented compactly through calls to *constraint abstractions*. In a companion paper [GS01] we show how to efficiently compute a least solution to constraints with constraint abstractions and we use this technique to obtain an $O(n^3)$ inference algorithm for our usage analysis, where n is the size of the explicitly typed program.

Acknowledgements We would like to thank David Sands and Makoto Takeyama for comments on this paper and Karl-Filip Faxén, Jakob Rehof and Keith Wansbrough for discussions on the relations to their work.

References

[BJ96] U. Boquist and T. Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *Proc. of IFL'96, Bad Godesberg, Germany.* Springer Verlag LNCS 1268, 1996.

[BS96] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6:579–612, 1996.

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *proceedings of 2nd Static Analysis Symposium*, September 1995.

[Fax95] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *Proc. of SAS'95*, pages 136–153. Springer-Verlag, LNCS 983, September 1995.

[Fax97] Karl-Filip Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs.* PhD thesis, Royal Institute of Technology, Sweden, June 1997.

[FW87] J. Fairbairn and S. Wray. TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators. In *Proc. of FPCA'87*, pages 34–45. Springer Verlag LNCS 274, September 1987.

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

[GS99] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *Proc. of HOOTS'99*, volume 26 of *ENTCS*. Elsevier, 1999.

[GS01] J. Gustavsson and J. Svenningsson. Constraint abstractions. In *Proc. of Second Symposium on Programs as Data Objects*, LNCS. Springer Verlag, 2001. To Appear.

[Gus98] J. Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. In *Proc. of ICFP'98*, pages 39–50, Baltimore, Maryland, September 1998.

[Gus99] J. Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. Licentiate thesis, May 1999.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, Charleston, N. Carolina, 1993.

[LGH$^+$92] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. L. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, Workshops in Computing, Glasgow, 1992.

[Mar93] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. In *Proc. 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer–Verlag, 1993.

[Mog97] T. Mogensen. Types for 0, 1 or many uses. In *Proc. of IFL '97*, pages 112–122. Springer-Verlag, LNCS 1467, September 1997.

[Mos97] C. Mossin. *Flow Analysis of Typed Higher-Order Programs (Revised Version)*. PhD thesis, University of Copenhagen, Denmark, August 1997.

[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, July 1992.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. of ICFP'96*, pages 1–12. ACM, May 1996.

[PS00] R. Peña and C. Segura. Non-determinism analysis in a parallel-functional language. In *Proceedings of the 12th International Workshop of Functional Languages*, LNCS, September 2000.

[RF01] Jakob Rehof and Manuel Fändrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of 2001 Symposium on Principles of Programming Languages*, 2001. To appear.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Sve00] Josef Svenningsson. An efficient algorithm for a sharing analysis with polymorphism and subtyping. Masters thesis, June 2000.

[TJ94] J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), 1994.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. of FPCA*, La Jolla, 1995.

[WPJ98] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. Technical Report TR-1998-19, Department of Computing Science, University of Glasgow, December 1998.

[WPJ99] Keith Wansbrough and Simon Peyton Jones. Once Upon a Polymorphic Type. In *Proc. of POPL'99*, January 1999.

[WPJ00] Keith Wansbrough and Simon Peyton Jones. Simple Usage Polymorphism. In *ACM SIGPLAN Workshop on Types in Compilation*, September 2000.