Usage Counting Analysis for Lazy Functional Languages

G. W. Hamilton

School of Computer Applications, Dublin City University,
Glasnevin, Dublin 9, Ireland

If it can be determined at compile-time how many times values will be used within lazy functional programs, a number of useful optimisations can be performed. For example, call-by-need parameter passing can be converted to call-by-name, and in-place updating and compile-time garbage collection can be performed. In this paper, it is shown how this usage counting information can be obtained by static analysis. This analysis is not itself a major contribution of this paper; similar analyses have been defined before. The major contributions of this paper are that it provides a framework against which this analysis can be proved correct for a lazy functional language, and the analysis is proved to be correct with respect to this framework. The framework for proving the correctness of the analysis is provided by defining a store semantics which counts the number of times values are used.

1. INTRODUCTION

Values within functional programs may be used several times. If it can be determined how many times they will be used, a number of useful optimisations can be performed. For example, if it can be determined that list cells will be used no more than once, then they can be annotated to indicate that they can be recycled after they are used. This optimisation is known as compile-time garbage collection (Hamilton, 1995).

Also, within lazy functional languages, the cost of checking whether a closure has been evaluated, and overwriting the closure with its value can be quite expensive. Usage counting information can be used to reduce this cost. If it is known that the result of an expression will be used at most once, the cost of checking whether the closure containing the expression has been evaluated and overwriting the closure with its value can be saved as it is not required.

Another possible application of usage counting analysis is to allow the in-place updating of large aggregate structures such as arrays (Hudak and Bloss, 1985). In
conventional functional implementations of arrays, the modification of an array involves making a copy of it, in case the original array is used again. If it can be determined that the original array is not used again, an in-place update of the array can be performed.

Usage counting analysis can also be beneficial when performing program transformation. If it can be determined that the arguments of a function will be used at most once, then calls of the function can be safely unfolded without reducing efficiency (Hamilton, 1993).

The usage information required to perform the above optimisations cannot be obtained through the application of strictness analysis. Strictness analysis determines whether values are used at least once. In order to perform the above optimisations, we need to know whether values are used at most once. A different analysis is therefore required.

In this paper, an analysis method is presented for determining the number of times values will be used within lazy functional programs. This analysis is not itself a major contribution of this paper; similar analyses have been defined before (Hudak, 1987; Hughes, 1988; Jensen and Mogensen, 1990; Jensen, 1990). The major contribution of this paper is that it proves this analysis to be correct with respect to a lazy functional language semantics.

Unlike strictness analysis, the usage counting analysis presented here cannot be proved correct with respect to the standard semantics of the language. The framework for proving the correctness of the static analysis is provided by defining a store semantics for the given language which counts the number of times values are used. This store semantics is shown to be congruent to the standard semantics of the language. The store semantics must be abstracted in some way to allow usage counts to be determined at compile-time. The usage counts within store values are therefore abstracted to usage patterns. These patterns are finite objects which indicate the number of times each part of a value is used. A usage counting analysis is then defined over these patterns to determine at compile-time the number of times each part of a value will be used in future computations. The usage pattern obtained by this analysis must be safe with respect to the actual usage count within the corresponding store value. This will be the case if the usage pattern of a value determined by the analysis is not less than the actual usage count, so it will not be incorrectly assumed that a value is no longer required by a program. The described usage counting analysis is proved to be safe with respect to the store semantics.

This paper summarises some of the work described in (Hamilton, 1993) and further details can be found there. The remainder of the paper is structured as follows. In Section 2, the syntax and standard semantics of the language on which the analysis is to be performed are defined. A store semantics is also defined for this language and this is shown to be congruent to the standard semantics. In Section 3, domains of usage patterns that are abstractions of the usage counts within store values are defined, along with the operations that can be performed upon them. In Section 4, a usage counting analysis is defined over the domains of usage patterns and some examples of its application are given. In Section 5, the usage counting analysis is proved to be correct with respect to the store semantics. In Section 6, related work is considered, and Section 7 concludes.
2. LANGUAGE: SYNTAX AND SEMANTICS

In this section, the syntax and semantics of the language that will be used throughout this paper are presented. The language is a simple first-order lazy functional language with list operators and recursion equations. To show that the usage counting analysis presented in this paper is correct, a reference must be provided against which its correctness can be proved. The standard semantic definition of the language is too abstract to provide usage information within a lazy functional language, so it cannot be used to provide this reference. A non-standard store semantics is therefore defined for the language and is shown to be congruent to the standard semantics. An alternative approach would be to define a lazy lambda calculus, as is done in (Launchbury, 1993) and (Ariola et al. 1995).

2.1. Notation

In this section, some of the notation that is used throughout this paper is described. It is assumed that the reader is familiar with domain theory. For a given domain $D$, the bottom element of the domain is represented by $\bot_D$, and the elements of the domain are ordered by the partial order $\sqsubseteq_D$. The notation $D_\perp$ represents the lifting of the domain $D$ to add a new bottom element $\perp$. The operators $\oplus$, $\times$ and $\rightarrow$ are the coalesced sum, product and function space constructors, respectively.

Tuples of elements are represented by $(v_1, ..., v_n)$. Elements of a tuple can be accessed using the $\downarrow$ operator, where $T \downarrow n$ denotes the $n$th element of the tuple $T$.

The notation $D^*$ represents zero or more function arguments which are elements of the domain $D$. Thus the function type $D^* \rightarrow E$ is a shorthand notation for $D \rightarrow \cdots \rightarrow D \rightarrow E$.

2.2. Syntax

In this section, the abstract syntax of the language which is used throughout this paper is defined. The language is a simple first order lazy functional language with list operators and recursion equations. The abstract syntax is shown in Fig. 1.

Programs in the language consist of an expression to evaluate and a set of function definitions. Nested function definitions are not allowed in the language. Programs involving nested function definitions can be transformed into this restricted form of program using a technique called lambda lifting (Johnsson, 1985). Some example function definitions are given in Fig. 2.

It is assumed that the language is monomorphically typed and that all programs in the language are well-typed. This restriction is necessary because the usage counting analysis presented later requires that each expression has a unique type. Values in the language can have the following types:

$$T ::= \text{int} \quad \text{Integers}$$

$$| \quad \text{bool} \quad \text{Booleans}$$

$$| \quad \text{list } T \quad \text{Lists}$$
The only constants in the abstract syntax of the language are integers. The basic functions are the built-in functions of the language and operate on integers only. The comparison of lists using the basic equality function is therefore not allowed, but it is possible to determine the equality of lists recursively within the language. Basic function applications will be expressed in infix notation throughout the course of this paper.

Booleans are represented by the values True and False. Note that booleans are considered to be constructors in the abstract syntax of the language. This is so that pattern matching can be performed upon them, since pattern matching is allowed only on constructors.

---

append xs ys = case xs of
  Nil    : ys
  Cons z xs : Cons z (append zs ys)

flatten xs = case xs of
  Nil    : Nil
  Cons zs xs : append xs (flatten zs)

reverse xs = case xs of
  Nil    : Nil
  Cons z xs : append (reverse xs) (Cons x Nil)

accreverse xs ys = case xs of
  Nil    : ys
  Cons z xs : accreverse xs (Cons x ys)

FIG. 1. Abstract syntax.

FIG. 2. Example function definitions.
The conditional can therefore be expressed as:

\[
\text{case } e_0 \text{ of } \\
\quad \text{True : } e_1 \\
\quad \text{False : } e_2.
\]

This has the same meaning as the more traditional form of conditional:

\[
\text{if } e_0 \text{ then } e_1 \text{ else } e_2.
\]

Empty lists are represented by \texttt{Nil} and non-empty lists are represented by an expression of the form \texttt{Cons e}_1 e_2, where the head of the list is denoted by \texttt{e}_1, and the tail of the list is denoted by \texttt{e}_2. Lists are decomposed using a \texttt{case} expression of the form

\[
\text{case } e_0 \text{ of } \\
\quad \texttt{Nil : } e_1 \\
\quad \texttt{Cons v}_1 v_2 : e_2.
\]

In the expression \(e_2\), the head of the list \(e_0\) is represented by the variable \(v_1\), and the tail of this list is represented by the variable \(v_2\). There is therefore no need to add explicit head and tail operators to the basic functions. Within \texttt{case} expressions of the form

\[
\text{case } e_0 \text{ of } \\
\quad p_1 : e_1 \\
\quad \vdots \\
\quad p_k : e_k.
\]

\(e_0\) is called the \texttt{selector}, and \(p_1 : e_1, \ldots, p_k : e_k\) are called the \texttt{branches}. Within the example expressions given throughout the course of this paper, the branches in a \texttt{case} expression are separated either by the | character or by a newline character. The patterns used in the branches of \texttt{case} expressions may not be nested. Methods to transform \texttt{case} expressions with nested patterns into ones without nested patterns are described in (Augustsson, 1985) and (Wadler, 1987).

The intended evaluation mechanism for the language is lazy evaluation. However, the basic functions are strict in all their arguments. Also, when a \texttt{case} expression is evaluated, the selector is evaluated to weak head normal form before the appropriate branch of the \texttt{case} expression is evaluated.

2.3. Standard Semantics

In this section a standard semantics is defined for the language that is used throughout this paper. The standard semantic domains are shown in Fig. 3.
Expressible values in the language are atomic values or lists. Atomic values consist of integers and booleans. Integers are represented by the flat domain of integers, and booleans are represented by the values TRUE and FALSE. Empty lists are represented by the value NIL and non-empty lists are represented by pairs, where the first element of the pair represents the head of the list, and the second element of the pair represents the tail of the list.

The functionality of the standard semantic functions of the language is shown in Fig. 4. \( \delta_p \) gives the meaning of a program, \( \delta \) gives the meaning of an expression, \( \beta_e \) gives the meaning of a basic function call, and \( \gamma_e \) gives the meaning of a constructor application. The function \( \text{match} \) is an auxiliary function which is used to perform pattern matching within \text{case} expressions. These functions are defined in Fig. 5. Empty environments are represented by \((\lambda x. \bot)\) in these functions, and non-empty environments are represented by \([x_1/v_1, \ldots, x_n/v_n]\), where each variable \(v_i\) is bound to the value \(x_i\). The notation \(\rho_e[x_1/v_1, \ldots, x_n/v_n]\) represents an environment in which each variable \(v_i\) is bound to the value \(x_i\), and all other variables are bound to the value given in the environment \(\rho_e\). Rather than give the definition of each basic function, a generic function \(b\) is used. This function is assumed to be an infix function with two arguments. For the sake of clarity, the domain injections and projections have been omitted from the semantics. These will be omitted from the semantics throughout the course of this paper, unless there is an ambiguity.

\[
\begin{align*}
\delta_p : & \quad \text{Prog} \rightarrow \text{Val}_e \\
\delta : & \quad \text{Exp} \rightarrow \text{Bve}_e \rightarrow \text{Fve}_e \rightarrow \text{Val}_e \\
\beta_e : & \quad \text{Bas} \rightarrow \text{Val}_e \rightarrow \text{Val}_e \\
\gamma_e : & \quad \text{Con} \rightarrow \text{Val}_e \rightarrow \text{Val}_e \\
\text{match} : & \quad (\text{Val}_e \times \text{Con}) \rightarrow \text{Bool}
\end{align*}
\]

FIG. 4. Standard semantic functions.
2.4. Store Semantics

In order to show that the usage counting analysis presented in this paper is correct, a reference must be provided against which its correctness can be proved. The standard semantic definition of the language presented in the previous section is too abstract to provide usage information, so it cannot be used to provide this reference. In this section, a non-standard semantics is presented which models the use of store. This non-standard semantics is largely based on the store semantics for a higher order lazy language presented in (Hughes, 1991), but has been augmented to incorporate usage counting. This involves counting the number of times each value is used in a program. The lazy store semantics described in (Hughes, 1991) is not used as a reference against which to prove the correctness of store-related analyses as is done in the work described here.
The store semantic domains are shown in Fig. 6. Most of these domains are similar to the domains for the standard semantics of the language given in Fig. 3, but some new domains have been added. Obviously, a domain of stores is required since the use of stores is being modeled. A store is represented by a function which returns the contents of a cell at a given location. Locations in the store are represented by integers. Unbound cells in the store are represented by the value UNB. Since the side-effect of updating a store is being modeled within the semantics, the current state of the store is threaded through the semantics. Values in the semantics are therefore represented by a pair, the first element of which is a location and the second a store.

As in the standard semantics, expressible values in the language are atomic values or lists. Atomic values consist of integers and booleans. Integers are represented by the flat domain of integers, and booleans are represented by the values TRUE and FALSE. Empty lists are represented by the value NIL, and non-empty lists are represented by pairs of locations that give the head and the tail of the list, respectively.

Within a lazy store semantics, it must be ensured that values are evaluated only when needed, and they are not evaluated more than once. A new domain of closures is therefore introduced for the store semantics. These closures are used to delay the evaluation of expressions until they are actually required by the program. They are represented by functions which, when supplied with a store, will return the result of evaluating their associated expression in the given store. The arguments of basic

\[
\begin{align*}
\text{Val}_\mathcal{S} & = (\text{Loc} \times \text{Store}_\mathcal{S})_\bot, \\
x \in \text{Eval} & = \text{Atom} \oplus \text{List} \\
\text{Atom} & = \text{Int} \oplus \text{Bool} \\
\text{Int} & = \{0\}_\bot \oplus \{1\}_\bot \oplus \{-1\}_\bot \oplus \ldots \\
\text{Bool} & = \{\text{TRUE}\}_\bot \oplus \{\text{FALSE}\}_\bot \\
\text{List} & = \{\text{NIL}\}_\bot \oplus \text{ConsCell} \\
\text{ConsCell} & = (\text{Loc} \times \text{Loc})_\bot \\
\text{loc} \in \text{Loc} & = \text{Int} \\
\text{Uval} & = (\text{Use} \times \text{Eval})_\bot \\
\eta \in \text{Use} & = \text{Int} \\
\text{Closure} & = \text{Store}_\mathcal{S} \rightarrow \text{Val}_\mathcal{S} \\
\rho_{\mathcal{S}} \in \text{Bvec}_\mathcal{S} & = \text{Bv} \rightarrow \text{Loc} \\
\phi_{\mathcal{S}} \in \text{Fvec}_\mathcal{S} & = \text{Fv} \rightarrow \text{Loc*} \rightarrow \text{Store}_\mathcal{S} \rightarrow \text{Val}_\mathcal{S} \\
\sigma_{\mathcal{S}} \in \text{Store}_\mathcal{S} & = \text{Loc} \rightarrow (\text{Closure} \oplus \text{Loc} \oplus \text{Uval} \oplus \{\text{UNB}\}_\bot)
\end{align*}
\]
function applications and selectors of case expressions are evaluated to weak head normal form because they appear in a strict context. All other expressions are enclosed within closures to delay their evaluation until their values are required by the program. For example, consider the evaluation of the expression square \((2 + 3)\), where the function square is defined as \(\text{square } x = x \times x\). The argument \((2 + 3)\) is not evaluated on evaluation of this function call, but is enclosed within a closure and allocated in the store. This closure must be supplied with a store before it can be evaluated. During the evaluation of the body of the function square, this closure will be evaluated only when the variable \(x\) is evaluated. This will occur when the arguments of the basic function application are evaluated. At this stage, the closure is applied to the current store.

To ensure that closures are not evaluated more than once, they are overwritten with the result of their evaluation immediately after they have been evaluated. For example, consider the evaluation of the expression square \((2 + 3)\) shown above. When the variable \(x\) within the body of the function square is used for the first time as an argument in a basic function application, its value is given by a closure containing the expression \((2 + 3)\). This closure is evaluated by applying it to the current store. The result of evaluating the closure is placed in the same location as the closure itself had been stored, thus overwriting it. The next time the variable \(x\) is used, also as an argument in a basic function application, its value is given by its previously evaluated result, so the expression \((2 + 3)\) does not need to be evaluated again.

Since the result of evaluating a closure is given by a location, cells in the store may contain the location of another cell in the store, but there are no chains of indirection. Also, since it must be possible to overwrite the closures given by bound variables with the result of their evaluation, variables in the bound variable environment are bound to locations. These locations will either be bound to a closure, or to another location if the closure has been evaluated.

Each expressible value in the semantics is allocated in the store. This is not necessary to ensure lazy evaluation, but is done so that a usage count can be associated with each expressible value. These usage counts are represented by integers. All new values which are created within a program are given an initial

\[
\begin{align*}
\mathcal{S}_p : & \quad \text{Prog} \rightarrow \text{Val}_p \\
\mathcal{F} : & \quad \text{Exp} \rightarrow \text{Bve}_p \rightarrow \text{Fve}_p \rightarrow \text{Store}_p \rightarrow \text{Val}_p \\
\mathcal{B}_p : & \quad \text{Bas} \rightarrow \text{Loc}_p^* \rightarrow \text{Store}_p \rightarrow \text{Val}_p \\
\mathcal{C}_p : & \quad \text{Con} \rightarrow \text{Loc}_p^* \rightarrow \text{Store}_p \rightarrow \text{Val}_p \\
\text{alloc} : & \quad ((\text{Closure} \odot \text{Uval}) \times \text{Store}_p) \rightarrow \text{Val}_p \\
\text{inc} : & \quad \text{Val}_p \rightarrow \text{Val}_p \\
\text{force} : & \quad \text{Val}_p \rightarrow \text{Val}_p \\
\text{match} : & \quad (\text{Eval} \times \text{Con}) \rightarrow \text{Bool}
\end{align*}
\]

FIG. 7. Store semantic functions.
usage count of 0 since they have not yet been used. These usage counts are incremented only when their associated values are used. This will be the case if a value appears in a strict context. The usage count for a value is therefore incremented only if it is an argument in a basic function call, a selector in a case expression, or its value is being forced as the result of a program. Usage counts can only increase as they are never decremented. As an example, consider the evaluation of the expression \( \text{square} \,(2 + 3) \) shown earlier. When the variable \( x \) within the body of the function \( \text{square} \) is used for the first time, the result of its evaluation is given

\[
\mathcal{S}\left[\begin{array}{c}
\text{where} \\
\text{f}_1 \ldots \text{f}_k = e_1 \\
\vdots
\end{array}\right] \\
\mathcal{S}\left[\begin{array}{c}
\text{f}_j \text{v}_1 \ldots \text{v}_{n_j} = e_j \\
\vdots
\end{array}\right] = \text{force}(\phi) \lambda \, \text{v}_1 \ldots \text{v}_{n_j} \, \text{loc} \, \text{v}_1 \ldots \text{v}_{n_j} \, \phi \, \sigma \, \text{f}_j \\
\text{where} \\
\phi = \mathbf{fix}(\lambda \phi \psi \phi \, \text{loc} \, \text{v}_1 \ldots \text{v}_{n_j} \, \phi \, \psi \, \sigma) / \text{f}_j \\
\mathcal{S}[k] \, \rho \, \psi \, \sigma = \text{alloc}(0, k, \psi \, \sigma) \\
\mathcal{S}[v] \, \rho \, \psi \, \sigma = (\text{loc} \, \phi \psi \, \text{loc} / \phi \psi \, \psi), \text{if} \, (\sigma \, \rho \, \psi \, \phi \psi \, \psi) \in \text{Closure} \\
\text{where} \\
(\text{loc} \, \phi \psi \, \psi) = (\sigma \, \rho \, \psi \, \phi \psi \, \psi) \, \sigma \, \psi \\
= ((\sigma \, \rho \, \psi \, \phi \psi \, \psi), \sigma \, \psi), \text{otherwise} \\
\mathcal{S}[b \, e_1 \ldots e_n] \, \rho \, \psi \, \sigma = \mathcal{S}[b] \, \text{loc} \ldots \text{loc} \, \sigma \\
\text{where} \\
(\text{loc} \, \phi \psi \, \phi \psi \, \sigma) = \text{inc}(\mathcal{S}[e_1] \, \rho \, \psi \, \sigma) \\
\vdots \\
(\text{loc} \, \phi \psi \, \phi \psi \, \sigma) = \text{inc}(\mathcal{S}[e_n] \, \rho \, \psi \, \phi \psi \, \sigma^{-1}) \\
\mathcal{S}[c \, e_1 \ldots e_n] \, \rho \, \psi \, \sigma = \mathcal{S}[c] \, \text{loc} \ldots \text{loc} \, \sigma \\
\text{where} \\
(\text{loc} \, \phi \psi \, \phi \psi \, \sigma) = \text{alloc}(\mathcal{S}[e_1] \, \rho \, \psi \, \sigma) \\
\vdots \\
(\text{loc} \, \phi \psi \, \phi \psi \, \sigma) = \text{alloc}(\mathcal{S}[e_n] \, \rho \, \psi \, \phi \psi \, \sigma^{-1}) \\
\mathcal{S}[f \, e_1 \ldots e_n] \, \rho \, \psi \, \sigma = \phi \psi \, \sigma / \text{loc} \ldots \text{loc} \, \sigma \\
\text{where} \\
(\text{loc} \, \phi \psi \, \phi \psi \, \sigma) = \text{alloc}(\mathcal{S}[e_1] \, \rho \, \psi \, \phi \psi \, \sigma) \\
\vdots \\
(\text{loc} \, \phi \psi \, \phi \psi \, \sigma) = \text{alloc}(\mathcal{S}[e_n] \, \rho \, \psi \, \phi \psi \, \sigma^{-1}) \\
\mathcal{S} \text{case} \ e_0 \ \text{of} \ \text{p}_1 : e_1 \ldots \text{p}_n : e_n \ | \ \rho \, \psi \, \sigma \\
= \mathcal{S}[e_0] \, \rho \, \psi \, \phi \psi \, \sigma \\
\text{where} \\
(\text{loc} \, \phi \psi \, \sigma) = \text{inc}(\mathcal{S}[e_0] \, \rho \, \psi \, \sigma) \\
(\text{u}, \text{x}) = \phi \psi \, \text{loc} \, \text{p}_1 \\
= \text{c} \, \text{v}_1 \ldots \text{v}_n \text{and match(x,c)} \\
\mathcal{B}[\text{v}] = \text{loc} \, \text{loc} \, \lambda \phi \psi \phi \psi \text{alloc}(0, \text{loc} \, \text{v}_1 \, \text{v}_2), \sigma \, \psi \\
\text{where} \\
(\text{u}_1, \text{u}_2) = \sigma \, \phi \psi \, \text{loc} \, \text{v}_1 \\
(\text{u}_2, \text{u}_2) = \sigma \, \phi \psi \, \text{loc} \, \text{v}_2

\text{FIG. 8.} \ \text{Store} \ \text{semantics.}
an initial usage count of 0. This is immediately incremented to 1 because it is an argument in a basic function call. When the variable \( x \) is used again, the usage count of its value is incremented to 2 because it is again an argument in a basic function call.

The functionality of the store semantic functions of the language is shown in Fig. 7. \( \mathcal{S} \) gives the meaning of a program and \( \mathcal{S}^\prime \) gives the meaning of an expression. The location returned by \( \mathcal{S} \) will be bound to an expressible value in the associated store. \( \mathcal{S}^\prime \) gives the meaning of a basic function application, and \( \mathcal{G}^\prime \) gives the meaning of a constructor application. These functions are defined in Figs. 8 and 9. The function \( \text{alloc} \) is used to allocate a given value at a location in the given store which was previously unbound. Both closures and expressible values can be allocated in this way. The function \( \text{inc} \) is used to increment the usage count associated with an expressible value. The function \( \text{force} \) is used to force the evaluation of the result of a program. It is possible that the result of a program contains closures. Closures which are reachable from the result must therefore be evaluated. When \( \text{force} \) is applied to a closure, it causes the evaluation of the closure. The result

\[
\mathcal{S} [\text{True}] = \lambda \sigma_x. \text{alloc}(0, \text{TRUE}, \sigma_x)
\]

\[
\mathcal{S} [\text{False}] = \lambda \sigma_x. \text{alloc}(0, \text{FALSE}, \sigma_x)
\]

\[
\mathcal{S} [\text{Nil}] = \lambda \sigma_x. \text{alloc}(0, \text{NIL}, \sigma_x)
\]

\[
\mathcal{S} [\text{Cons}] = \lambda \text{loc}_1, \text{loc}_2, \lambda \sigma_x. \text{alloc}(0, (\text{loc}_1, \text{loc}_2), \sigma_x)
\]

\[
\text{alloc}(v, \sigma_x) = (\text{loc}, \sigma_x[\text{loc}/\text{loc}])
\]

where

\[
\sigma_x \text{ loc} = \text{UNB}
\]

\[
\text{inc}(\text{loc}, \sigma_x) = (\text{loc}, \sigma_x[(u + 1, x)/\text{loc}])
\]

where

\[
(u, x) = \sigma_x \text{ loc}
\]

\[
\text{force}(\text{loc}, \sigma_x) = (\text{loc}, \sigma_x), \quad \text{if } (\sigma_x \text{ loc}) = \bot \text{ or } (\sigma_x \text{ loc}) = \text{UNB}
\]

\[
= (\text{loc}', \sigma_x'[\text{loc'}/\text{loc}]), \quad \text{if } (\sigma_x \text{ loc}) \in \text{Closure}
\]

where

\[
(\text{loc}', \sigma_x') = \text{force}(\sigma_x \text{ loc}, \sigma_x)
\]

\[
= \text{force}(\sigma_x \text{ loc}, \sigma_x), \quad \text{if } (\sigma_x \text{ loc}) \in \text{Loc}
\]

\[
= \text{inc}(\text{loc}, \sigma_x'[\text{loc}, \text{loc}_2]/\text{loc}), \quad \text{if } x \in \text{Consecc}
\]

where

\[
(u, x) = \sigma_x \text{ loc}
\]

\[
(\text{loc}, \sigma_x) = \text{force}(x \downarrow 1, \sigma_x)
\]

\[
(\text{loc}_2, \sigma_x) = \text{force}(x \downarrow 2, \sigma_x)
\]

\[
= \text{inc}(\text{loc}, \sigma_x), \quad \text{otherwise}
\]

\[
\text{match}(x, c) = (x = \text{TRUE} \text{ and } c = \text{True})
\]

\[
\text{or } (x = \text{FALSE} \text{ and } c = \text{False})
\]

\[
\text{or } (x = \text{NIL} \text{ and } c = \text{Nil})
\]

\[
\text{or } (x \in \text{Consecc} \text{ and } c = \text{Cons})
\]

Fig. 9. Store semantics (continued).
of this evaluation is also forced. When it is applied to a list value it is recursively
applied to the elements of the list, forcing their evaluation. All other values that can
result from the evaluation of a program will have been fully evaluated already, and
do not need to be forced. It is assumed that this function also serves to print out
the result of the program. The function \textit{match} is used to perform pattern matching
within \texttt{case} expressions as before.

2.5. Congruence

Since the store semantics of the language will be used as a reference against
which the usage counting analysis can be proved correct, the store semantics and
standard semantics of the language must be shown to be congruent. A function \( \Phi \)
is therefore defined which is used to extract the standard semantic component from
a store value. The store semantics and standard semantics of the language can then
be shown to be congruent if the results of evaluating programs in both semantics
have the same standard semantic component. The definition of the function \( \Phi \)
and the following proof are similar to a function and proof given in (Hughes,

\begin{definition}
(Standard semantic component of a store value). The standard
semantic component of a store value can be extracted using the function \( \Phi \) which
is defined as follows:

\[
\Phi: \text{Val}_{\text{\sigma}} \rightarrow \text{Val}_{\text{\epsilon}}
\]

\[
\Phi(\text{loc}, \sigma_{\text{\epsilon}}) = \bot, \quad \text{if} \quad (\sigma_{\text{\epsilon}} \text{ loc}) = \bot \quad \text{or} \quad (\sigma_{\text{\epsilon}} \text{ loc}) = \text{UNB}
\]

\[
= \Phi((\sigma_{\text{\epsilon}} \text{ loc}) \sigma_{\text{\epsilon}}), \quad \text{if} \quad (\sigma_{\text{\epsilon}} \text{ loc}) \in \text{Closure}
\]

\[
= \Phi((\sigma_{\text{\epsilon}} \text{ loc}) \sigma_{\text{\epsilon}}), \quad \text{if} \quad (\sigma_{\text{\epsilon}} \text{ loc}) \in \text{Loc}
\]

\[
= (\Phi(x \downarrow 1, \sigma_{\text{\epsilon}}), \Phi(x \downarrow 2, \sigma_{\text{\epsilon}})), \quad \text{if} \quad x \in \text{Conscell}
\]

\[
\quad \text{where}
\]

\[
(u, x) = \sigma_{\text{\epsilon}} \text{ loc}
\]

\[
= x, \quad \text{otherwise}
\]

\[
\quad \text{where}
\]

\[
(u, x) = \sigma_{\text{\epsilon}} \text{ loc}
\]

\end{definition}

This function forces the evaluation of any closures in the store value and extracts
the standard semantic component from the resulting value. Using this definition,
the congruence of expressions in the store semantics and standard semantics of the
language can be shown by proving the following lemma.
Lemma 2.2. (Congruence of expressions).

\( \forall e : \text{Exp}, \quad \phi : \text{Fve}_{\mathcal{E}}, \quad \sigma : \text{Store}_{\mathcal{E}}, \quad \phi_{\mathcal{E}} : \text{Fve}_{\mathcal{E}} \): \\
\text{if } \forall f \in \text{dom}(\phi) : \\
\Phi(\phi_{\mathcal{E}}[f] \mid \text{loc}_1, \ldots, \text{loc}_n, \sigma) = \phi_{\mathcal{E}}[f](\Phi(\text{loc}_1, \sigma), \ldots, (\Phi(\text{loc}_n, \sigma))) \\
\text{then } \forall e : \text{Exp}, \quad \forall v \in \text{dom}(\rho) : \\
\Phi(\mathcal{E}[e] \mid \rho, \phi_{\mathcal{E}}[v]) = \delta[v][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}]

Proof. The proof is by structural induction on the expression \( e \).

Base cases

Case 1. \( e ::= k \)

\[ \Phi(\mathcal{E}[k] \mid \rho_{\mathcal{E}}[v]) = k \]

\[ \Rightarrow \Phi(\mathcal{E}[k] \mid \rho_{\mathcal{E}}[v]) = \delta[k][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}] \]

Case 2. \( e ::= v \)

\[ \Phi(\mathcal{E}[v] \mid \rho_{\mathcal{E}}[v]) = \Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}) \]

\[ \Rightarrow \Phi(\mathcal{E}[v] \mid \rho_{\mathcal{E}}[v]) = \delta[v][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}] \]

Inductive cases

Case 1. \( e ::= b e_1 \ldots e_n \)

\[ \Phi(\mathcal{E}[b e_1 \ldots e_n] \mid \rho_{\mathcal{E}}[v]) = \delta[b][\Phi(\mathcal{E}[e_1] \mid \rho_{\mathcal{E}}[v]), \ldots, \Phi(\mathcal{E}[e_n] \mid \rho_{\mathcal{E}}[v]), \phi_{\mathcal{E}}] \]

\[ = \delta[b][\delta[e_1][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}]), \ldots, \delta[e_n][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}]) \]

(by inductive hypothesis)

\[ = \delta[b e_1 \ldots e_n][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}]) \]

\[ \Rightarrow \Phi(\mathcal{E}[b e_1 \ldots e_n] \mid \rho_{\mathcal{E}}[v]) = \delta[b e_1 \ldots e_n][\Phi(\rho_{\mathcal{E}}[v], \sigma_{\mathcal{E}}), \phi_{\mathcal{E}}]) \]

Case 2. \( e ::= c e_1 \ldots e_n \)

\[ \Phi(\mathcal{E}[c e_1 \ldots e_n] \mid \rho_{\mathcal{E}}[v]) = \delta[c][\Phi(\mathcal{E}[e_1] \mid \rho_{\mathcal{E}}[v]), \ldots, \Phi(\mathcal{E}[e_n] \mid \rho_{\mathcal{E}}[v]), \phi_{\mathcal{E}}]) \]
= C[c](δ[e₁][Φ(ρ, τ, στ)[ψ] ϕ])...
(δ[eₙ][Φ(ρ, τ, στ)[ψ] ϕ])...
(by inductive hypothesis)
= δ[c e₁...eₙ][Φ(ρ, τ, στ)[ψ] ϕ] ϕₙ
⇒ Φ(ς[e₁...eₙ] ρ, ψ, στ) = δ[e₁...eₙ][Φ(ρ, τ, στ)[ψ] ϕ] ϕₙ
Case 3.  e ::= fe₁...eₙ
Φ(ς[f e₁...eₙ] ρ, ψ, στ) = ϕₙ[f] Φ(ς[e₁] ρ, ψ, στ)...
Φ(ς[eₙ] ρ, ψ, στ) = ϕₙ[f] (δ[e₁][Φ(ρ, τ, στ)[ψ] ϕ] ϕ)...
(δ[eₙ][Φ(ρ, τ, στ)[ψ] ϕ] ϕ)...
(by inductive hypothesis)
⇒ Φ(ς[f e₁...eₙ] ρ, ψ, στ) = δ[f e₁...eₙ][Φ(ρ, τ, στ)[ψ] ϕ] ϕₙ
Case 4.  e ::= case e₀ of p₁: e₁ |...| pₖ: eₖ
Φ(ς[case e₀ of p₁: e₁ |...| pₖ: eₖ] ρ, ψ, στ) = Φ(ς[e₁] ρ, x₁,..., xₙ/τ₁,..., τₙ)[ψ] σ')
where
(loc, α') = Φ[e₀] ρ, φ σ' = x = α' loc
pᵢ = c v₁...vₙ and match(x, c)
= δ[e₁][ϕₚ[x₁,..., xₙ/τ₁,..., τₙ] ϕₚ]...
(by inductive hypothesis)
= δ[e₀ of p₁: e₁ |...| pₖ: eₖ][Φ(ρ, τ, στ)[ψ] ϕ] ϕₙ
The following lemma states that the functional variable environments in the store semantics and standard semantics of the language will always satisfy the requirement in Lemma 2.2.

**Lemma 2.3. (Congruence of functional variable environments).**

∀ρ ∈ Prog:
if ⌦ρ[ρ] = ⌦[e](λx.⊥) ϕ
and ⌦ρ[ρ] = force(⌟[e](λx.⊥) ϕ(λloc. UNB))
then ∀v ∈ dom(ϕ, σ, ρ) ∈ Store:
Φ(ϕ[ρ][f] loc1...locn, σ, ρ) = ϕ[ρ][f](Φ(loc1, σ, ρ))...(Φ(locn, σ, ρ))

**Proof.** The proof of this lemma is by fixed point induction.

**Base case.** The first approximations to each function variable environment are as follows:

ϕ0 = ([x1...xk, ⊥]/f]
ϕ0 = ([x1...xk, ⊥]/f]

Φ(ϕρ[f] loc1...locn, σ, ρ) = ⊥
= ϕρ[f](Φ(loc1, σ, ρ))...(Φ(locn, σ, ρ))

**Inductive case**

ϕρ[f] loc1...locn, σ, ρ

where f is defined by f, vj1...vjk = e

Φ(ϕρ[f] loc1...locn, σ, ρ)

(by inductive hypothesis and Lemma 2.2)

= ϕρ[f](Φ(loc1, σ, ρ))...(Φ(locn, σ, ρ))

The congruence of programs in the store semantics and standard semantics of the language can now be shown by proving the following theorem.
Theorem 2.4. (Congruence of programs).

\[ \forall p \in \text{Prog}: \Phi(\mathcal{F}_p[\![p]\!] ) = \ell_p[\![p]\!] \]

Proof. This theorem follows immediately from Lemmata 2.2 and 2.3.

3. USAGE PATTERNS

The usage counting store semantics defined in the previous section must be abstracted in some way to allow usage counts to be determined at compile-time. The approach that is taken here is to abstract usage counting store values to usage patterns that represent the number of times each part of a value is used in future computations. The future usage of a value is also called its context.

The notation \( D_{\text{ABS}} \) used in the definition of the usage counting domains represents the lifting of the domain \( D \) to add a new bottom element \( \text{ABS} \). The element \( \text{ABS} \) represents absence, indicating that an expression is not evaluated, so no parts of it are used. This lifting operation is defined as follows.

Definition 3.1. (The domain lifting operation).

\[
D_{\text{ABS}} = D \cup \{ \text{ABS} \}
\]

where

\[
\begin{align*}
\text{ABS} & \subseteq D_{\text{ABS}} d, \quad \forall d \in D_{\text{ABS}} \\
d_1 & \subseteq D_{\text{ABS}} d_2, \quad \forall d_1, d_2 \in D \quad \text{s.t.} \quad d_1 \subseteq_d d_2
\end{align*}
\]

A different domain of usage patterns is defined for each possible type of value in the language. The domain of usage patterns for a value of type \( T \) is given by \( U(T) \). The type \( T' \) in the definition of the domain \( U(T') \) represents an atomic type (\( \text{int, bool} \)). These domains are defined as follows.

Definition 3.2. (Domains of Usage Patterns).

\[
U(T') = (U'(T'))_{\text{ABS}}
\]

\[
U'(T') = \{ 1, 2 \}
\]

where

\[
1 \subseteq_u U'(T') 2
\]

\[
U(\text{list } T) = (U'(\text{list } T))_{\text{ABS}}
\]

\[
U'(\text{list } T) = (U'(T') \times U(T))
\]

where

\[
(u_1, u_2) \subseteq_u U'(\text{list } T) (u'_1, u'_2), \quad \text{if} \quad u_1 \subseteq_u U'(T') u'_1 \quad \text{and} \quad u_2 \subseteq_u U(T) u'_2
\]
Each domain \( U(T) \) is an abstract context domain as defined in (Hughes, 1998), with the least element ABS representing absence. There is no element in any of the domains \( U(T) \) representing contradiction because it is assumed that all programs are well typed, and contradiction can never arise.

Elements of the domain \( U(T_\alpha) \) describe the usage of values of atomic type. The elements in this domain, other than ABS, are the usage patterns 1 and 2, which indicate that a value is used at most once or may be used any number of times respectively.

Elements of the domain \( U(list \ T) \) describe the usage of list values containing elements of type \( T \). This domain is similar to the finite abstract context domain for lists defined in (Hughes, 1988). Elements of this domain, other than ABS, are pairs, where the first element of the pair describes the usage of all the spine cells in the list (these consist of the root cell of the list together with root cells of the lists obtained by successively taking the tail of the list), and the second element describes the usage of all the elements in the list. Since these elements describe the usage of more than one value, they give a safe approximation to the usage of all of them. The usage of the spine cells of a list are represented by the values 1 and 2. The value 1 indicates that none of the spine cells are used more than once, and the value 2 indicates that all the spine cells may be used any number of times. The usage of the elements in the list are described by the usage domain corresponding to their type.

Each domain \( U(T) \) is a complete lattice, with the least element representing absence, and the greatest element representing a value in which all parts may be used any number of times. The domain \( U(list \ T_\alpha) \) can be viewed as shown in Fig. 10.

We now define the operations which can be performed upon usage patterns. When the usage of a value in one expression is given by \( u_1 \), and the usage of the

---

![Diagram](attachment:image.png)

**FIG. 10.** The domain of usage patterns \( U(list \ T_\alpha) \).
same value in another expression is given by $u_2$, a means of combining these two usage patterns into one describing the total usage of the value in both expressions is required. As in (Hughes, 1988), a binary operator $\&$ is defined to provide this information. This operator can be regarded as an abstract addition operator over elements in each domain $U(T)$. It is defined on the domain of usage patterns for values of atomic type as follows.

**Definition 3.3 (The $\&$ operator).**

\[
\begin{align*}
    u \& \text{ABS} &= u, & \forall u \in U(T_A) \\
    u \& 1 &= 1, & \text{if } u = \text{ABS} \\
    & = 2, & \text{otherwise} \\
    u \& 2 &= 2, & \forall u \in U(T_A)
\end{align*}
\]

The definition of this operator is extended pointwise on domains of usage patterns for values of structured type. For example, if the usage of a variable of type `list int` within one expression is given by $(1, 1)$, and the usage of the variable within another expression is given by $(2, \text{ABS})$, then the total usage of the variable in both expressions is given by $(2, 1)$.

The two usage patterns which are combined using this operator will be safe approximations to the usage of a value in two different expressions. The usage pattern which is produced as the result of this operator will therefore be a safe approximation to the total usage of the value in both expressions, since it simply acts as an abstract addition operator over domains of usage patterns.

Also following (Hughes, 1988), the binary operator $\rightarrow$ is defined to preserve absence in the context ABS. It is defined for each domain of usage patterns as follows.

**Definition 3.4 (The $\rightarrow$ operator).**

\[
\begin{align*}
    u_1 \rightarrow u_2 &= \text{ABS}, & \text{if } u_1 = \text{ABS} \\
    & = u_2, & \text{otherwise}
\end{align*}
\]

If an expression appears in the context ABS, then no part of the result of the expression will be used, and so no part of the sub-expressions occurring within it will be used either. It must therefore be ensured that any absence in the context of an expression is propagated to all sub-expressions. For example, if the list `Cons x y` appears in the context ABS, then the usage of $x$ and $y$ will also be given by ABS.

To determine the usage of a constructor application from the usage of its arguments, abstract constructors which operate on usage patterns are defined. Corresponding to each constructor $c$ of type $T_1 \rightarrow \ldots \rightarrow T_n$, abstract constructors $\forall c$ which are of type $U(T_A) \rightarrow U(T_1) \rightarrow \ldots \rightarrow U(T_n)$ are defined as follows.
Definition 3.5 (The abstract constructors \( \forall c \)).

\[
\begin{align*}
\forall \text{False}(u_0) &= u_0 \\
\forall \text{True}(u_0) &= u_0 \\
\forall \text{Nil}(u_0) &= (u_0, \text{ABS}) \\
\forall \text{Cons}(u_0, u_1, u_2) &= (u_0, u_1) \sqcup u_2
\end{align*}
\]

The binary operator \( \sqcup \) gives the least upper bound of two usage patterns in each domain of usage patterns. The additional argument for each abstract constructor is an element of the usage domain \( U(T_A) \). It represents the usage of the overall resulting structure if it is of atomic type, or the usage of the root cell of the resulting structure if it is of list type. The usage of the spine cells in a list is given by the least upper bound of the usage of the root cell of the list, and the usage of the spine cells in the tail of the list. The usage of the elements in a list is given by the least upper bound of the usage of the head of the list and the usage of the elements in the tail of the list. For example, if the root cell of the list \( \text{Cons} x y \) is used once, the usage of \( x \) is given by \( 2 \), and the usage of \( y \) is given by \( (2, 1) \), then the usage of the overall list is given by \( \forall \text{Cons}(1, 2, (2, 1)) = (1, 2) \sqcup (2, 1) = (2, 2) \).

The usage of the head and tail of a list can be determined from the usage of the overall list using the \( \forall \text{Cons}^1 \) and \( \forall \text{Cons}^2 \) operators, respectively. These operators are defined as follows:

Definition 3.6 (The \( \forall \text{Cons}^1 \) and \( \forall \text{Cons}^2 \) operators).

\[
\begin{align*}
\forall \text{Cons}^1 \text{ABS} &= \text{ABS} \\
\forall \text{Cons}^1(u_1, u_2) &= u_2 \\
\forall \text{Cons}^2 \text{ABS} &= \text{ABS} \\
\forall \text{Cons}^2(u_1, u_2) &= (u_1, u_2)
\end{align*}
\]

For example, if the usage of the list \( \text{Cons} x y \) is given by \( (2, 1) \), then the usage of \( x \) is given by \( \forall \text{Cons}^1(2, 1) = 1 \), and the usage of \( y \) is given by \( \forall \text{Cons}^2(2, 1) = (2, 1) \).

Now that the operations on usage patterns have been defined, it remains to prove that they are monotonic and continuous. The proofs are not difficult, and are not included here.

4. USAGE COUNTING ANALYSIS

In this section, a usage counting analysis is presented which operates over the domains of usage patterns. This analysis determines the maximum number of times a value will be used in future computations within a program. The functionality of the usage counting analysis functions is shown in Fig. 11.
Each function in the function variable environment in the analysis gives the future usage of a given argument within a given function for a given context of function call. The function $\mathcal{U}$ gives the function variable environment resulting from the usage counting analysis of a program. The future usage (or context) of a value of type $T$ is an element of the usage domain $U(T)$ and is represented by $u$ in this analysis. The result of evaluating $U[e][x] \phi_f$ gives the maximum number of times the variable $x$ is used in future computations if the expression $e$ appears in the context $u$. These functions are defined in Fig. 12. The rules for this analysis can be explained as follows:

(U1) The result of evaluating a program is a function variable environment in which functions of the form $Uf^k$ are introduced. Each function of the form $Uf^k$ gives the future usage of argument number $k$ within the function $f$ for a given context of function call. The value of this function variable environment is determined using a least fixed point evaluation.

(U2) No part of a variable is used in a constant.

(U3) If the variable $x$ is evaluated in a context $u$, then the usage of $x$ is given by $u$. If any other variable is evaluated, then the variable $x$ is absent.

(U4) Each of the arguments in a basic function application will be evaluated in a context 1, since they will be used only once. The total usage of the variable $x$ is the total (using $\&$) of its usage in each of these arguments.

(U5) If a constructor application is evaluated in a context $u$, then each of its arguments will be evaluated in a context given by the sub-component of $u$ which corresponds to that argument. The total usage of the variable $x$ is the total (using $\&$) of its usage in each of these arguments.

(U6) If a function application is evaluated in a context $u$, then each of its arguments will be evaluated in a context given by the function variable environment for a call of the function in the context $u$. The total usage of the variable $x$ is the total (using $\&$) of its usage in each of these arguments.

(U7) If a case expression is evaluated in a context $u$, then the branches of the case expression will also be evaluated in the context $u$. The context in which
the selector of the case expression will be evaluated depends upon which branch of
the expression is selected. This context is given by the application of the abstract
constructor (corresponding to the constructor in the pattern of the selected branch)
to the usage patterns giving the usage of the pattern matching variables in the
selected branch. The total usage of the variable \( x \) in the case expression is the total

\[
\text{TABLE 1 Using Counting Analysis of the Function } \text{append}
\]

<table>
<thead>
<tr>
<th>Context</th>
<th>( (1, \text{ABS}) )</th>
<th>( (1, 1) )</th>
<th>( (1, 2) )</th>
<th>( (2, \text{ABS}) )</th>
<th>( (2, 1) )</th>
<th>( (2, 2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{append #1} )</td>
<td>( \text{ABS} )</td>
<td>( (1, \text{ABS}) )</td>
<td>( (1, 1) )</td>
<td>( (1, 2) )</td>
<td>( (1, \text{ABS}) )</td>
<td>( (1, 1) )</td>
</tr>
<tr>
<td>( \text{append #2} )</td>
<td>( \text{ABS} )</td>
<td>( (1, \text{ABS}) )</td>
<td>( (1, 1) )</td>
<td>( (1, 2) )</td>
<td>( (2, \text{ABS}) )</td>
<td>( (2, 1) )</td>
</tr>
</tbody>
</table>
TABLE 2

Usage Counting Analysis of the Function reverse

<table>
<thead>
<tr>
<th>Context</th>
<th>ABS</th>
<th>(1, ABS)</th>
<th>(1, 1)</th>
<th>(1, 2)</th>
<th>(2, ABS)</th>
<th>(2, 1)</th>
<th>(2, 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#reverse #1</td>
<td>ABS</td>
<td>(1, ABS)</td>
<td>(1, 1)</td>
<td>(1, 2)</td>
<td>(1, ABS)</td>
<td>(1, 1)</td>
<td>(1, 2)</td>
</tr>
</tbody>
</table>

The results of applying usage counting analysis to the function append are shown in Table 1. From this table, it can be seen that the spine cells in the first argument of append will never be used more than once, and the list elements in the first argument will be used the same number of times as the list elements in the result of the function. The usage of the second argument of append will be exactly the same as the usage of the result of the function.

The results of applying the analysis to the function reverse are shown in Table 2. From this table, it can be seen that the spine cells in the argument of reverse will never be used more than once, and the list elements in the argument will be used the same number of times as the list elements in the result of the function.

The results of applying the analysis to the function accreverse are shown in Table 3. From this table, it can be seen that the spine cells in the first argument of accreverse will never be used more than once, and the list elements in the first argument will be used the same number of times as the list elements in the result of the function. The usage of the second argument of accreverse will be exactly the same as the usage of the result of the function.

The results of applying the analysis to the function flatten are shown in Table 4. From this table, it can be seen that no list cells in the argument of flatten will ever be used more than once, and the bottom level elements in each list in the argument
TABLE 4

Usage Counting Analysis of the Function flatten

<table>
<thead>
<tr>
<th>Context</th>
<th>(1, ABS)</th>
<th>(1, 1)</th>
<th>(1, 2)</th>
<th>(2, ABS)</th>
<th>(2, 1)</th>
<th>(2, 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flatten</td>
<td>(1, (1, ABS))</td>
<td>(1, (1, 1))</td>
<td>(1, (1, 2))</td>
<td>(1, (1, ABS))</td>
<td>(1, (1, 1))</td>
<td>(1, (1, 2))</td>
</tr>
</tbody>
</table>

will be used the same number of times as the list elements in the result of the function.

The usage counting analysis defined above allows us to determine the usage of a variable within an expression, given the usage of the overall expression. We need to know whether the usage of the overall expression is safe for all the possible contexts in which this expression may appear during evaluation of the program. An algorithm is therefore required to assign a safe usage count to each expression within a program. If a safe context $u$ can be determined for each function, then a safe usage count can be determined for each sub-expression $e$ within the function body $e'$ by replacing $e$ with the variable $v$ within $e'$ and determining $U[e'][v] u \phi u$. It remains to determine a safe context for each function. Two different approaches can be taken. The first is to take the least upper bound of all the contexts in which the function appears. This is done in (Jensen, 1990) by using a collecting semantics. The second approach is to take the top element of the appropriate usage counting domain, which is guaranteed to be safe. This is the approach which is taken here. It has the advantage of avoiding the inefficiency of a collecting semantics, but the disadvantage of being less precise. We argue that this is accurate enough to give good results in many cases. For example, consider the reverse function given in Fig. 2. If this function is of type list int, and we want to determine the usage of the recursive function call (reverse xs), the calculation would be as shown in Fig. 13.

FIG. 13. Example application of usage counting analysis.
As can be seen, the spine cells of the expression (reverse xs) will be used only once. It is therefore possible to optimise this expression to indicate that all its spine cells can be recycled after their first use. This will reduce the number of Cons cells allocated for the function reverse from $O(n^2)$ for an argument list of length $n$, to $O(n)$. It is also possible to reduce the number of Cons cells allocated for sorting functions from $O(n^2)$ for an argument list of length $n$, to $O(n)$.

5. PROOF OF CORRECTNESS

Since the information obtained from usage counting analysis is going to be used to allow various optimisations to be performed, it must be shown that it is safe with respect to the usage counting store semantics. This will be the case if the future usage of a value obtained by usage counting analysis is a safe approximation to the increment in usage of the value in the usage counting store semantics due to the evaluation of the program. It will be a safe approximation if it is greater than or equal to the actual usage.

To determine the usage pattern corresponding to the increment in usage of a usage counting store value, the function $\delta$ is defined as follows.

**Definition 5.1 (Usage pattern corresponding to the increment in usage of a usage counting store value).** The usage pattern corresponding to the increment in usage of a usage counting store value at location $loc$ between the stores $\_\_\_\_$ can be determined for each type of value using the function $\delta$ which is defined as follows:

\[
\delta: (\text{Loc}_t \times \text{Store}_t \times \text{Store}_t) \rightarrow U(T) \\
\delta(loc, \sigma, \sigma') = \text{ABS, } \begin{cases} \\
\text{if } (\sigma loc) = \text{UNB or } (\sigma loc) = \bot \\
\text{or } ((\sigma' loc) \downarrow 1) = ((\sigma loc) \downarrow 1) \\
\text{= } \delta((\sigma loc), \sigma, \sigma'), \text{ if } (\sigma loc) \in \text{Loc} \\
\text{= } u, \text{ otherwise} \\
\end{cases} \\
\text{where} \\
u = 1, \text{ if } ((\sigma' loc) \downarrow 1) - ((\sigma loc) \downarrow 1) = 1 \\
u = 2, \text{ otherwise} \\
\delta: (\text{Loc}_t \times \text{Store}_t \times \text{Store}_t) \rightarrow U(\text{list} T) \\
\delta(loc, \sigma, \sigma') = \text{ABS, } \begin{cases} \\
\text{if } (\sigma loc) = \text{UNB or } (\sigma loc) = \bot \\
\text{or } ((\sigma' loc) \downarrow 1) = ((\sigma loc) \downarrow 1) \\
\text{= } \delta((\sigma loc), \sigma, \sigma'), \text{ if } (\sigma loc) \in \text{Loc} \\
\text{= } \text{Nil}(u_0), \text{ if } (\sigma loc) \downarrow 2 = \text{NIL} \\
\text{where} \\
u_0 = 1, \text{ if } ((\sigma' loc) \downarrow 1) - ((\sigma loc) \downarrow 1) = 1 \\
u_0 = 2, \text{ otherwise} \\
\text{Nil}(u_0, u_1, u_2), \text{ if } (\sigma loc) \downarrow 2 \in \text{Conscell} \\
\text{where} \\
u_0 = 1, \text{ if } ((\sigma' loc) \downarrow 1) - ((\sigma loc) \downarrow 1) = 1 \\
u_0 = 2, \text{ otherwise} \\
\]
The function $\delta$ can be viewed as an abstract subtraction operator on the usage counts of store values. It is assumed that all closures have been evaluated before the usage pattern corresponding to the increment in usage of a usage counting store value is determined. If the value at the given location in the store is unbound or is undefined, or there is no increment in the usage of the value, then the corresponding usage pattern is ABS. If there is an increment of one in the usage of an atomic value, then the corresponding usage pattern is 1, otherwise it is 2. The usage pattern corresponding to a list value is determined recursively from the store value and gives the least upper bound of the usage of the spine cells of the list, and the least upper bound of the usage of the elements in the list.

In order to prove the correctness of the usage counting analysis, one would expect to show that if the usage of an expression $e$ is given by $u$, then the usage of each variable $x$ within $e$ is less than or equal to $U[e][x] u \phi_w$. This would be stated as follows:

$$
\text{if} \quad S[p] = (\text{loc}^\sigma, \sigma^\pi) \\
\text{and} \quad S[e \mid x_i] \phi \sigma = (\text{loc}', \sigma') \\
\text{and} \quad \delta(\text{loc}', \sigma', \sigma^\pi) \subseteq u \\
\text{then} \quad \delta(\text{loc}', \sigma', \sigma^\pi) \subseteq (S[e \mid x_i] u \phi_w)
$$

This is not possible because parts of each variable $x$ within the expression $e$ may be shared, and may be used within other expressions. We therefore show that if the usage of an expression $e$ is given by $u$, and the usage of each variable $x$ within $e$ is made greater than or equal to $U[e][x] u \phi_w$, then the usage of the expression $e$ must be greater than or equal to $u$. This ensures that the usage of the variables within the expression as obtained by usage counting analysis is greater than or equal to the actual usage. The correctness of the usage counting analysis can therefore be shown by proving the following theorem.

**Theorem 5.2 (Correctness of usage counting analysis).**

For all $p, \varphi \in \text{Bve}_w, \phi, \psi \in \text{Fve}_w, \sigma, \sigma' \in \text{Store}_w, \phi \psi \in \text{Fve}_w, p \in \text{Prog}, e \in \text{Exp}:

- if $S[p] = (\text{loc}^\sigma, \sigma^\pi)$
- and for all $f \in \text{dom}(\phi)$:
  - if $\phi[f] \text{loc}_1 \ldots \text{loc}_n \sigma = (\text{loc}', \sigma')$
  - and $\delta(\text{loc}', \sigma', \sigma^\pi) = u$
  - then if $\phi[f] \text{loc}_1 \ldots \text{loc}_n \sigma = (\text{loc}', \sigma')$
    - and $(\phi \psi \text{[\#]} u) \subseteq \delta(\text{loc}', \sigma', \sigma^\pi)$
    - then $u \subseteq \delta(\text{loc}', \sigma', \sigma^\pi)$
- and $S[e] \rho \psi \phi \sigma = (\text{loc}', \sigma')$
- and $\delta(\text{loc}', \sigma', \sigma^\pi) = u$
- then for all $x_i \in \text{dom}(\rho_w)$:
if $\mathcal{S}[e][loc_i/x_i] \phi \sigma_{\phi} = (loc^*, \sigma_{\phi}^*)$
and $(\forall[e][x_i] u \phi \psi) \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$
then $u \subseteq \delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*)$

Proof. The proof of this theorem is by structural induction.

Base cases

Case 1. $e ::= k$

if $\mathcal{S}[p] = (loc^m, \sigma_{\phi}^m)$
and $\mathcal{S}[e] \rho \phi \sigma_{\psi} = (loc^*, \sigma_{\psi}^*)$
and $\delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*) = u$
then if $\mathcal{S}[e][loc_i/x_i] \phi \sigma_{\phi} = (loc^*, \sigma_{\phi}^*)$
and $(\forall[e][x_i] u \phi \psi) \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$
then $\delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*) = u$
(since no part of $x_i$ appears in the result of $e$)

$\Rightarrow u \subseteq \delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*)$

Case 2. $e ::= v$

if $\mathcal{S}[p] = (loc^m, \sigma_{\phi}^m)$
and $\mathcal{S}[e] \rho \phi \sigma_{\psi} = (loc^*, \sigma_{\psi}^*)$
and $\delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*) = u$
then if $\mathcal{S}[e][loc_j/x_i] \phi \sigma_{\phi} = (loc^*, \sigma_{\phi}^*)$
and $\forall[e][x_i] u \phi \psi \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$
then $u \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$, if $v = x_i$,
and $\forall \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$, otherwise

$\Rightarrow u \subseteq \delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*)$, if $v = x_i$
and $\delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*) = u$, otherwise
(since no part of $x_i$ appears in the result of $e$)

$\Rightarrow u \subseteq \delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*)$

Inductive cases

Case 1. $e ::= b e_1 \ldots e_n$

if $\mathcal{S}[p] = (loc^m, \sigma_{\phi}^m)$
and $\mathcal{S}[e] \rho \phi \sigma_{\psi} = (loc^*, \sigma_{\psi}^*)$
and $\delta(loc^*, \sigma_{\phi}^*, \sigma_{\psi}^*) = u$
and $\mathcal{S}[e][loc_j/x_i] \phi \sigma_{\phi} = (loc^*, \sigma_{\phi}^*)$
and $\forall[e][x_i] u \phi \psi \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$
then $(u \rightarrow (\forall[e_1][x_i] 1 \phi \psi \& \ldots \& \forall[e_n][x_i] 1 \phi \psi)) \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$
$\Rightarrow$ if $\mathcal{S}[e][loc_j/x_i] \phi \sigma_{\phi} = (loc^*, \sigma_{\phi}^*)$
then $\forall[e][x_i] 1 \phi \psi \subseteq \delta(loc_i, \sigma_{\phi}, \sigma_{\psi}^*)$, if $u \neq \forall$
126 G. W. HAMILTON

\[ \Rightarrow 1 \subseteq \delta(\text{loc}', \sigma', \sigma'_w), \text{ if } u \neq \text{ABS} \]
(by inductive hypothesis)

\[ \Rightarrow u \subseteq \delta(\text{loc}'', \sigma'', \sigma''_w) \]

Case 2. \[ e ::= e_1 \ldots e_n \]

if \[ \mathcal{S}_e[p] = (\text{loc}'', \sigma''_w) \]
and \[ \mathcal{S}_e[p] \rho \sigma \mathcal{S}_e = (\text{loc}', \sigma'_w) \]
and \[ \delta(\text{loc}', \sigma', \sigma'_w) = u \]
and \[ \mathcal{S}_e[\text{loc}'', x_1] \rho \sigma \mathcal{S}_e = (\text{loc}'', \sigma''_w) \]
and \[ (\mathcal{S}_e[x_1] u \mathcal{S}_e) \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]
then \[ u \rightarrow (\mathcal{S}_e[x_1] u_1 \mathcal{S}_e \ldots u_n \mathcal{S}_e) \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]

\[ \Rightarrow u \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]
(by inductive hypothesis)

\[ u \subseteq \delta(\text{loc}'', \sigma'', \sigma''_w) \]

Case 3. \[ e ::= f e_1 \ldots e_n \]

if \[ \mathcal{S}_e[p] = (\text{loc}'', \sigma''_w) \]
and \[ \mathcal{S}_e[p] \rho \sigma \mathcal{S}_e = (\text{loc}', \sigma'_w) \]
and \[ \delta(\text{loc}', \sigma', \sigma'_w) = u \]
and \[ \mathcal{S}_e[\text{loc}'', x_1] \rho \sigma \mathcal{S}_e = (\text{loc}'', \sigma''_w) \]
and \[ (\mathcal{S}_e[x_1] u \mathcal{S}_e) \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]
then \[ u \rightarrow (\mathcal{S}_e[x_1] u_1 \mathcal{S}_e \ldots u_n \mathcal{S}_e) \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]

\[ \Rightarrow u \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]
(by inductive hypothesis)

\[ u \subseteq \delta(\text{loc}'', \sigma'', \sigma''_w) \]

(by assumptions for \( \mathcal{S}_e \) and \( \mathcal{S}_e \) in Theorem 5.2)

Case 4. \[ e ::= \text{case } e_0 \text{ of } p_1:: e_1 | \ldots | p_k:: e_k \]

if \[ \mathcal{S}_e[p] = (\text{loc}'', \sigma''_w) \]
and \[ \mathcal{S}_e[p] \rho \sigma \mathcal{S}_e = (\text{loc}', \sigma'_w) \]
and \[ \delta(\text{loc}', \sigma', \sigma'_w) = u \]
and \[ \mathcal{S}_e[\text{loc}'', x_1] \rho \sigma \mathcal{S}_e = (\text{loc}'', \sigma''_w) \]
and \[ (\mathcal{S}_e[x_1] u \mathcal{S}_e) \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]
then \[ u \rightarrow ((\mathcal{S}_e[x_1] u_1 \mathcal{S}_e) \ldots (\mathcal{S}_e[x_1] u_n \mathcal{S}_e)) \ldots \]
\[ \Delta((\mathcal{S}_e[x_1] u_k \mathcal{S}_e) \ldots (\mathcal{S}_e[x_1] u_n \mathcal{S}_e)) \subseteq \delta(\text{loc}', \sigma', \sigma''_w) \]

(by inductive hypothesis)
if $p$ is the branch $p_j$; $e_j$ is selected

and $S[e][x_i] \phi_{e_j} \sigma_{e_j} = (loca', \sigma'_e)$

and $S[e][x_i] \phi_{e} \sigma_{e} = (loca, \sigma'_e)$

then $\forall [e][x_i] u_j \phi(e) \subseteq \delta(loca, \sigma_e, \sigma'_e)$

and $\forall [e][x_i] u \phi(e) \subseteq \delta(loca, \sigma_e, \sigma'_e)$

$u \subseteq \delta(loca", \sigma_e, \sigma'_e)$

(by inductive hypothesis)

The following lemma states that the function variable environment in the usage counting analysis will satisfy the requirement in Theorem 5.2. This will be the case if the usage of a function call is given by $u$ and the usage of each argument within the function call is made greater than or equal to $\phi[f] u$, then the usage of the function call is greater than or equal to $u$. This ensures that the usage of the arguments within the function call as obtained by usage counting analysis is greater than or equal to the actual usage.

**Lemma 5.3 (Correctness of function variable environment).**

for all $p \in \text{Prog}$:

if $p[p] = \text{force}(S[e](loca, \bot) \phi_e (loca, \text{UNB})) = (loca", \sigma'_e)$

and $\forall p[p] = \phi_e$

then for all $f \in \text{dom}(\phi_e)$, $\sigma_e \in \text{Store}_e$

if $\phi_e[f] loca_1...loca_n \sigma_e = (loca', \sigma'_e)$

and $\delta(loca', \sigma_e, \sigma'_e) = u$

then if $\forall [f] loca'_1...loca'_n \sigma_e = (loca", \sigma'_e)$

and $\forall (\phi_e[f] u) \subseteq \delta(loca', \sigma_e, \sigma'_e)$

then $u \subseteq \delta(loca", \sigma_e, \sigma'_e)$

**Proof.** The proof of this lemma is by recursion induction.

**Base case**

if $p[p] = (loca", \sigma'_e)$

and $\phi_e[f] loca_1...loca_n \sigma_e = (loca', \sigma'_e)$

then $S[e][loca_1/v_1, ... , loca_n/v_n] \phi_{e} \sigma_{e} = (loca', \sigma'_e)$

where $f_j$ is defined by $f_j v_1...v_n = e_j$

if $\delta(loca', \sigma_e, \sigma'_e) = u$

then if $\phi_{e} loca'_1...loca'_n \sigma_e = (loca", \sigma'_e)$

and $\forall (\phi_e[\#k] u) \subseteq \delta(loca'_n, \sigma_e, \sigma'_e)$

then $\forall [e][loca'_1/v_1, ... , loca'_n/v_n] \phi_{e} \sigma_{e} = (loca", \sigma'_e)$

and $\forall (\phi[e][v_n] u \phi_{e}) \subseteq \delta(loca'_n, \sigma_e, \sigma'_e)$

$u \subseteq \delta(loca", \sigma_e, \sigma'_e)$

(by Theorem 5.2, since the function $f$ is not recursive)
Inductive case

$$\phi_{\sigma}^{n+1} = \left[ (\lambda u. U[e_j][v_{j_1} \ldots v_{j_m}] u \phi_{\sigma}^{n}) \downarrow f_j \# k \right]$$
where $$f_j$$ is defined by $$f_j v_{j_1} \ldots v_{j_m} = e_j$$

$$\phi_{\sigma'}^{n+1} = \left[ (\lambda \text{loc}_1 \ldots \text{loc}_{j_m} \lambda \sigma_{\sigma'}. U[e_j][\text{loc}_i v_j v_j \ldots \text{loc}_{j_m} v_{j_m}] \phi_{\sigma'}^{n} \sigma_{\sigma'})/f_j \right]$$
where $$f_j$$ is defined by $$f_j v_{j_1} \ldots v_{j_m} = e_j$$

\[
\begin{align*}
\text{if} & \quad \sigma'[p] = (\text{loc}_n, \sigma_{\sigma'}) \\
\text{and} & \quad \phi_{\sigma'}^{n+1}[f_j] \text{loc}_1 \ldots \text{loc}_{j_m} \sigma_{\sigma'} = (\text{loc}_c', \sigma_{\sigma'}) \\
\text{then} & \quad \sigma'[e_j][\text{loc}_j v_{j_1} \ldots \text{loc}_{j_m} v_{j_m}] \phi_{\sigma'}^{n} \sigma_{\sigma'} = (\text{loc}_c', \sigma_{\sigma'}) \\
\text{if} & \quad \delta(\text{loc}_c', \sigma_{\sigma'}, \sigma_{\sigma'}) = u \quad \Rightarrow \\
& \quad \Rightarrow u \subseteq \delta(\text{loc}^*, \sigma_{\sigma'}, \sigma_{\sigma'})
\end{align*}
\]
(by inductive hypothesis and Theorem 5.2)

6. APPLICABILITY AND PRACTICALITY OF THE ANALYSIS

The applicability and practicality of usage counting analysis have not yet been considered. This concerns how the analysis can be applied to real programs, and how practical this will be.

6.1. Applicability

The usage counting analysis presented earlier is applicable to a first-order monomorphic language in which there are no user-defined data types. However, real functional programs have user-defined data types, polymorphism and higher order functions. In this section, each of these is considered in turn, and it is shown how usage counting analysis can be extended to cope with them.

First of all, consider user-defined data types. If a user-defined data type $$T$$ has component types $$T_1 \ldots T_n$$, then the usage of a value of type $$T$$ can be described by the domain $$U(T, \{ T_i \})$$ where $$U(T, s)$$ is defined as follows:

\[
\begin{align*}
U(T, s) &= (U'(T, s))_{\text{ABS}} \\
U'(T, s) &= (U'(T_d) \times U(T_1, s \cup \{ T_1 \}) \times \ldots \times U(T_n, s \cup \{ T_n \}))
\end{align*}
\]

where

\[
\begin{align*}
\{ T_1, \ldots, T_n \} &= \{ T | T \in \{ T_1, \ldots, T_n \} \land T \not\equiv s \} \\
(u_0, u_1, \ldots, u_k) \subseteq U'(T_d) & \quad \text{if } u_0 \subseteq U'(T_d), u_0 \\
\text{and } u_1 \subseteq U(T_1, s \cup \{ T_1 \}) u_1, & \\
\ldots & \\
\text{and } u_k \subseteq U(T_k, s \cup \{ T_k \}) u_k
\end{align*}
\]
The parameter $s$ within this definition is a set of the types which have have been encountered before during the definition of the usage domain. If one of these types is encountered again, its usage domain is not added again to the overall usage domain being defined. This ensures that all the usage domains defined in this way are finite, as there must be a finite number of types within a program and their usage domain cannot be added more than once.

The $\&$ and $\sqcup$ operators can be extended pointwise to these usage domains, and the $\rightarrow$ operator can be defined as before. The definition of the abstract constructors $\forall c$ and the $\forall c \# i$ operators must also be extended to incorporate these new usage domains. If a constructor $c$ is of type $(T_1 \times \ldots \times T_n) \rightarrow T$ and $U'(T, s) = (U'(T_A) \times U(T_1', s \cup \{T_1\}) \times \ldots \times U(T_k', s \cup \{T_k\}))$, then the abstract constructor $\forall c$ is defined as follows.

$$ \forall c : (U'(T_A) \times U(T_1, s \cup \{T_1\}) \times \ldots \times U(T_n, s \cup \{T_n\})) \rightarrow U(T, s) $$

$$ \forall c(u_0, u_1, \ldots, u_n) = u' \sqcup (u'_1, \ldots, u'_k) $$

where

$$ u' = \bigsqcup_{i=1}^n \{ u_i | u_i \in U(T, s) \} $$

$$ u'_1 = \bigsqcup_{i=1}^n \{ u_i | u_i \in U(T_1', s \cup \{T_1\}) \} $$

$$ \vdots $$

$$ u'_k = \bigsqcup_{i=1}^n \{ u_i | u_i \in U(T_k', s \cup \{T_k\}) \} $$

If a constructor $c$ is of type $(T_1 \times \ldots \times T_n) \rightarrow T$ and $U'(T, s) = (U'(T_A) \times U(T_1', s \cup \{T_1\}) \times \ldots \times U(T_k', s \cup \{T_k\}))$, then the operators $\forall c \# i$ where $i \in \{1\ldots n\}$ are defined as follows.

$$ \forall c \# i : U(T, s) \rightarrow U(T_i, s \cup \{T_i\}) $$

$$ \forall c \# i \text{ ABS} = \text{ABS} $$

$$ \forall c \# i(u_0, u_1, \ldots, u_k) = (u_0, u_1, \ldots, u_k), \quad \text{if } T_i = T $$

$$ = u_j, \quad \text{if } T_i = T'_j \quad \text{where } j \in \{1\ldots k\} $$

Now consider polymorphism. If a polymorphic type checker is used, it is possible that the type of an expression cannot be determined precisely. For example, a polymorphic type checker will determine that the elements of the list argument of the reverse function shown in Fig. 2 can be of any type. This is because the list elements are not used in evaluating the result of the function. It is shown in (Abramsky, 1985) that if an analysis using abstract interpretation is a polymorphic invariant (gives the same results for every instance of a polymorphic type),
then the analysis can be applied to the simplest instance, and the result used for all instances. As stated in (Hughes, 1988), it is likely that a similar result also holds for backward analysis. It is straightforward to show that usage counting analysis is polymorphically invariant, as a sub-expression will be of polymorphic type only if it is not used in evaluating the result of the surrounding expression which was type checked. The usage domain for values of atomic type is therefore sufficient for describing the usage of values of polymorphic type.

Finally, the extension of usage counting analysis to include higher order functions is considered. The work described in (Hughes, 1988) sketches an extension of backward analysis to deal with higher order functions. This is done by combining the backward analysis with abstract interpretation, where the abstract function domain maps the context of a function application onto the context propagated to the argument. A similar extension could also be applied to the analysis described in this paper.

A sketch is also given in (Jensen and Mogensen, 1990) and (Jensen, 1990) of an extension to a usage counting analysis similar to the one described in this paper to deal with higher order functions. This involves using a closure analysis like the one described in (Sestoft, 1989) to determine the set of possible abstract closures to which a function can be evaluated during the execution of a program. The least upper bound of the corresponding contexts of these abstract closures is then determined. A global environment is represented by a grammar as before, and an approximation to this grammar is determined at compile-time.

6.2. Practicality

The usage counting analysis described in this paper has been implemented only for a toy language, so there is no experimental evidence that this approach is practical within the context of a real functional language. However, it is possible to determine the complexity of the analysis to get some idea of its practicality.

Within the usage counting analysis, it is necessary to calculate a fixed point in a domain of the form $U(T) \rightarrow U(T')$ for each context function. The size of this domain is dependent on the size of the types $T$ and $T'$. These domains are generally quite small, so the number of iterations required to find each fixed point will also be small. The number of context functions which need to be determined is dependent on the number of function arguments within a program. At most $O(\text{program size})$ iterations are therefore required to find all context functions, and since each iteration can be completed in time $O(\text{program size})$, the total time for a usage counting analysis is proportional to the square of the program size in the worst case. This is more efficient than other analyses which make use of abstract interpretation or type inference, which tend to be exponential in the worst case. As is the case for analyses which make use of abstract interpretation or type inference, the worst case for the usage counting analysis is likely to arise rarely.
7. RELATED WORK

In this section, other usage counting analyses, within the three frameworks of backward analysis, abstract interpretation and type inference, are considered.

7.1. Backward Analysis

Two simple backward analysis are described in (Hughes, 1988) which are relevant to the work described here. The first analysis can be used to determine usage counting information. The information obtained by this analysis can be used to optimise call-by-need to call-by-name, thus saving the cost of overwriting a closure with its value, and testing to see whether the overwrite has been performed. The domain used in this analysis contains contexts which are subsets of the set \(\{0,1,M\}\), whose elements mean that a value is used zero, one, or many times, respectively. The abstract addition operator \& is defined as \(a \& b = \{a + b | a \in \alpha \land b \in \beta\}\). The usage counting analysis in (Hughes, 1988) is defined only for values of atomic type, but a general method for extending contexts for atomic data to give contexts for sum types, product types and lists is given which could be applied to the usage counting analysis. No method for extending contexts for all data types is given. The usage counting domain in (Hughes, 1988) is larger than the one which is used in the work described here. This is because the domain in (Hughes, 1988) contains all possible combinations of usage counts within sets as elements, but in the work described here, these possible combinations are coalesced to give a single approximation, thus providing a smaller domain. The domain which is used in this work will therefore require less iterations in fixed point calculations. It is possible, however, to determine more information such as strictness using the domain in (Hughes, 1988).

The second analysis in (Hughes, 1988) which is relevant to the work described in this paper is to determine the life-time of data. Consider a function \(f\) which is defined as \(f x = g(hx)\). If \(h\) returns a data structure, it will be allocated in the store. If the value of \(h\) is used by \(g\) and is then discarded, it is possible to store the value of \(h\) in a short-term store which is discarded when the function \(f\) returns. For example, in a stack-based implementation, the value of \(h\) could be placed in the activation record for \(f\) instead of in the heap. However, as is pointed out in (Hughes, 1988), in a lazy language the result of \(h\) could be used only by \(g\), but might have a long life-time because a closure referring to it is not evaluated until long after \(f\) returns, so the use of short-term storage would be inappropriate. Also, the implementation of multiple short-term stores of different sizes and life-times presents a lot of problems.

A backward analysis for determining usage counting information for structured data is described in (Jensen and Mogensen, 1990) and (Jensen, 1990). This analysis is very similar to the usage counting analysis presented in this paper. The domain used in this analysis is defined as \(D = \{\text{ABS, 0}\} \cup \{n(d_1, d_2) | n \in \{1, 2\}, d_1, d_2 \in D\}\). A usage count of the form \(n(d_1, d_2)\) describes the usage of a list where the usage of the root cell is given by \(n\), and the usage of the head and tail of the list are given...
by $d_1$ and $d_2$, respectively. This domain therefore distinguishes between the sharing of each of the spine cells in a list. This differs from the approach taken in this paper, where one usage count is used to describe the usage of all the spine cells in a list. This is a reasonable approach to take since all the elements in a list are normally treated in the same way within a program. The domains described in this paper will therefore be smaller than that used in (Jensen and Mogensen, 1990) and (Jensen, 1990). Also, the domain $D$ is infinite, so the usual iterative method for finding fixpoints will not terminate in general. This situation is avoided by using a global environment which binds variables to the set of contexts in which they appear, and binds functions to the set of contexts of the calls to them. This global environment is represented by a grammar, and it is possible to determine an approximation to this grammar at compile-time. This makes the analysis even less efficient as it is defined on the power domain of the domain $D$ given above. No proof of correctness is given for the analysis described in (Jensen and Mogensen, 1990) and (Jensen, 1990).

7.2. Abstract Interpretation

Usage counting information can also be determined through the use of abstract interpretation. This has the disadvantage of being more costly to calculate a fixed point than it is within a backward analysis that is defined over a similar domain. This is because each abstract function within an abstract interpretation has $n$ arguments, whereas within a backward analysis they have one argument.

An isolation interpretation is described in (Mycroft, 1981) which can be used to determine if data structures are used no more than once in a strict first-order functional language. This extends previous work in (Schwarz, 1978) in which these isolation classes had to be supplied by the user. An approximate set of isolation patterns are determined for each value. This interpretation is relatively complex and makes use of information obtained by two other static analyses, the $E_{USES}$ interpretation and the $E_{EXAM}$ interpretation. The $E_{USES}$ interpretation is used to determine which parts of a list appear directly in the result of an expression. The $E_{EXAM}$ interpretation is used to determine which parts of a list are traversed when accessing substructures located deeper in the structure. No proof of correctness is given for the isolation interpretation.

The sharing analyses described in (Jones and Le Métayer, 1989) and (Hamilton and Jones, 1990) are applicable to strict first order functional languages and are similar to the isolation interpretation described in (Mycroft, 1981). They also make use of the information obtained by two other static analyses; transmission analysis and necessity analysis. These analyses are similar to the $E_{USES}$ and $E_{EXAM}$ interpretations described in (Mycroft, 1981). The domains of sharing patterns which are used in these analyses distinguish between the sharing of each of the spine cells in a list. To allow the compile-time analysis of sharing, these domains are cut off at a suitable depth. This differs from the approach taken in this paper where one usage count is used to describe the usage of all the spine cells in a list. The domains described in this paper will therefore be smaller than those described in (Jones and Le Métayer, 1989) and (Hamilton and Jones, 1990). The correctness of the sharing
analyses described in (Jones and Le Métayer, 1989) and (Hamilton and Jones, 1990) is not considered.

In (Hudak, 1987), an abstract interpretation is defined which models reference counting in a first-order strict functional language with one-dimensional arrays. The aim is to determine when in-place updating of an array can be performed instead of creating a new copy of an old one. This analysis involves counting the number of syntactic pointers to values within a program. This differs from usage counts, which are the total number of future accesses to a cell. A value may be referenced many times, but it might be used only once. The analysis uses an environment and a store. The environment maps variables to locations, and the store maps locations to (location, store) pairs. This is similar to the approach taken in the store semantics defined in this paper. To allow the analysis of reference counting at compile-time, “sticky” reference counts are used. When a reference count reaches a certain maximum value, it cannot be reduced again. A “collecting interpretation” of reference counts is then used to collect for each expression in a program all the possible reference counts the value of the expression can have. A proof of the safety of the analysis is also given. A similar analysis for a higher order strict language is described in (Andersen, 1990). The analyses presented in (Hudak, 1987) and (Andersen, 1990) use an abstract store and are therefore likely to be inefficient. Abstract stores tend to be relatively large objects, so in order to find a fixed point within such analyses, the values at all locations in the abstract store must be identical. A large number of iterations may therefore be required before a fixed point is reached.

The work described in (Inoue et al., 1988) and (Hughes, 1992) both make use of information obtained by an inheritance analysis and a generation analysis. The inheritance analysis is used to determine which cells will appear directly in the result of a function, and the generation analysis is used to determine which cells are created within a function argument. Cells generated within a function argument which are unshared and do not appear in the result of the function can be collected after evaluation of the function call. To determine whether generated cells are unshared, an overlapping analysis is presented in (Inoue et al., 1988). In (Hughes, 1991), it is observed that cells are always shared at the same level in a list in a well-typed language. A complete level of a list which is generated can therefore be explicitly deallocated in-masse if it is not inherited. This method cannot be used for lazy languages, since arguments which do not appear in the result of a function may not have been evaluated during the evaluation of the function. Attempting to explicitly deallocate these arguments may therefore force their evaluation, which is unsafe when using a lazy evaluation strategy. Another problem with this method of explicit deallocation is that there may be a substantial delay between a cell becoming garbage and its explicit deallocation. This is because cells are explicitly deallocated only after the evaluation of a function call. The need for run-time garbage collection will therefore not be delayed as long as possible. This is also a problem for the compile-time garbage collection methods described in (Mycroft, 1981), (Jones and Le Métayer, 1989) and (Hamilton and Jones, 1990).

An update avoidance analysis is presented in (Marlow, 1993), which can be used to determine the number of times a value will be used in future computations. If the
value is used no more than once, the cost of updating a closure with the result of its evaluation can be avoided. The analysis involves collecting a bag of variables which must be used when a given expression is evaluated. A bag is used because the same variable may be used more than once. The number of times a variable is used in evaluating the expression can then be determined by counting the number of occurrences of the variable in the bag. The analysis does not deal with structured data, and no proof of correctness is given for it.

7.3. Type Inference

Usage counting information can also be obtained through the use of type inference. This has a similar disadvantage to abstract interpretation in that the analysis can take time which is exponentially proportional to the size of the program in the worst case.

The update avoidance analysis described in (Launchbury et al., 1992) is a type scheme which can be used to determine usage counting information. This type scheme is defined on a domain similar to the usage counting domain presented in this paper for values of an atomic type, so it does not give very detailed usage counting information for structured data. The information obtained by the analysis is used to avoid updating a closure with its value, if its value is used only once. No correctness proof is given for the analysis because no appropriate semantics could be defined as a reference for its correctness.

A type inference scheme for usage counting analysis is also presented in (Wright and Baker-Finch, 1993). This scheme is based on relevant logic. It involves monitoring applications of the contraction structural rule to determine the number of times a value is used. The usage count of a value is incremented each time the contraction rule is applied to it. The described work does not give an algorithm for assigning types to terms. Also, it does not deal with data structures, and recursion is considered only informally.

Another type inference scheme for determining usage information is described in (Courtenage and Clack, 1994). This system can be used to determine whether a value is used zero times, exactly once, or at least once. It is not possible to determine whether a value is used at most once. This is the usage information which we are trying to determine in this paper.

The usage counting analysis presented in (Turner et al., 1995) also uses type inference to determine usage information. This scheme is proved to be correct with respect to the operational semantics described in (Launchbury, 1993) and the call-by-need calculus described in (Ariola et al., 1995). The described analysis is less precise than that presented here as absence is not detected. Also, if a list is used more than once, it is assumed that the elements of the list will be used more than once.

The type schemes described in (Wadler, 1990; Guzmán and Hudak, 1990; Barendsen and Smetsers, 1993) allow the user to indicate that a value will be used once. The linear type scheme described in (Wadler, 1990) is based on linear logic (Girard, 1987). Values which are declared to be linear in this type scheme must be
used exactly once. No distinction is made between sharing and absence. Some usage information is therefore lost. The type scheme described in (Guzmán and Hudak, 1990) is more loosely based on linear logic and can be used to determine that values are used no more than once. This type scheme is therefore not as restrictive as the linear type scheme described in (Wadler, 1990), but the type rules are considerably more complex. The unique type scheme described in (Barendsen and Smetsers, 1993) makes use of graph reduction information to determine whether values are unique. A value is unique if there is exactly one path to it from the graph root. This is a similar restriction to that in the linear type scheme described in (Wadler, 1990).

8. CONCLUSION

In this paper, an analysis method for determining the number of times values will be used within lazy functional programs has been presented. This analysis provides useful information for a compiler and allows a number of useful optimisations to be performed, thus increasing the efficiency of functional language implementations. The framework for proving the correctness of the analysis was provided by defining a store semantics for the language which counts the number of times values are used. The analysis was then proved to be correct with respect to this store semantics.

Received August 1, 1995; final manuscript received February 25, 1998

REFERENCES


