Types for 0, 1 or many uses

Torben Æ. Mogensen

DIKU
Universitetsparken 1
DK-2100 Copenhagen O
Denmark
email: torbenm@diku.dk
phone: (+45) 35321404
fax: (+45) 35321401

Abstract.
This paper will present an analysis for detecting single uses of values in functional programs during call-by-need reduction. There are several reasons why such information can be useful. The Clean language uses a uniqueness type system for detecting single-threaded uses which allow destructive updating. Single-use information has also been proposed for compile-time garbage collection. Turner, Wadler and Mossin have presented a single-use analysis which is intended to detect cases where call-by-need can safely be replaced by call-by-name.
This paper will focus on this last application of single-use analysis and present a type-based analysis which overcomes some limitations present in the abovementioned analysis.

Keywords: Program analysis, storage management, functional programming, types, constraints.

1 Introduction

This paper will present an analysis for detecting single uses of values in functional programs during call-by-need reduction. There are several reasons why such information can be useful. The Clean language [6] uses a uniqueness type system [1] for detecting single-threaded uses which allow destructive updating. In [3], single-use information is proposed for compile-time garbage collection using the observation that the any use of a single-use value will be the last use. In [10], Turner, Wadler and Mossin present an analysis based on linear types which is intended to detect cases where call-by-need can safely be replaced by call-by-name. The present paper will focus on this last application of single-use analysis and present a type-bases analysis which overcomes some limitations present in the analysis shown in [10].

The analysis is (like Turner *et al.*'s) based on a type system. We use the type rules to generate a set of constraints that are solved off-line using a linear-time constraint solver.

We do not attempt to prove our analysis correct, but it is expected that an approach similar to that used in [10] can be used.

1.1 Limitations in Turner *et al.*'s analysis

Turner *et al.*'s analysis can handle simple cases of zero usage by use of structural rules similar to those found in linear logic. Since this method involves adding or deleting entire variables in the environment, it cannot handle zero usage of parts of variables, only whole variables.

Additionally, there is a restriction on usage-annotated types that means that whenever the spine of a list can be used repeatedly, it is assumed that the same is true for the elements. As an example, the function

$$mean\ l = sum\ l\ /\ length\ l$$

traverses the spine of l twice, but accesses the lements of l only once. The analysis cannot detect this, as it assumes each spine-traversal implies access to the elements.

In this paper, we introduce a type-based analysis that will be able to detect single use in such cases by removing the restriction on structured types and by introducing the ability to detect zero uses of parts of values by means of an explicit 0 usage annotation on types.

2 The language

We use a simple functional language with integers, pairs and recursive datatypes and function definitions. This language is slightly larger than the language presented in [10], but this is mainly to show how general recursive data structures (*i.e.* more than lists) are handled.

2.1 Types

We assume that the programs are simply typed using the following types

$$t\ ::=\ Int \mid t_1 \times t_2 \mid \mu\alpha.c_1\ t_1 + \cdots + c_n\ t_n \mid \alpha \mid t_1 \to t_2$$

where α is a type variable used for recursive types. We will later discuss possible extension to polymorphic types.

2.2 Expressions

The abstract syntax of the language is given below. n is any natural number, the x_i are variables and the c_i are constructors.

$$\begin{array}{rl} e ::= & x \mid n \mid e_1 + e_2 \mid (e_1, e_2) \mid split\ e_1\ to\ (x_1, x_2)\ in\ e_2 \\ \mid & c_i\ e \mid case\ e_0\ of\ c_1\ x_1\ \to\ e_1; \ldots; c_n\ x_n\ \to\ e_n \\ \mid & e_1\ e_2 \mid \lambda x : t.e \mid let\ x : t = e_1\ in\ e_2 \\ \mid & letrec\ x_1 : t_1 = e_1, \ldots, x_n : t_n = e_n\ in\ e_0 \end{array}$$

Note the *split* expression which breaks down a pair into its components. We use this instead of *fst* and *snd* because these would require two uses of a pair to get at the components, where *split* uses only one. Hence, uses of *split* instead of *fst* and *snd* will give more cases of single use.

Pattern matching is naturally translated into uses of *case* and *split* (moreso than with *fst* and *snd*) and it is easy to define *fst* and *snd* in terms of split. Hence, it is no serious limitation to use *split*.

The *case* expressions are assumed to be exhaustive (though this doesn't really matter for the analysis).

3 Adding usage annotations

We will now annotate types and expressions with usage counts. The count indicates an upper bound on the usage of values of the annotated expression or type. Usage counts can be any of 0, 1 or ∞. A count of 0 indicates that the value is certain never to be used. While this isn't likely to be the case for a value at the place of its construction, zero-counts are used to express that a variable (or part thereof) is unused in a subexpression. 1 means *at most* one use. ∞ indicates that no upper bound on the use has been detected.

3.1 Annotated types

A count k on a type constructor in a well-typed program means than any value of that type is used at most k times during execution of the program.

When we use a type variable with a superscript count or count variable, e.g. t^1 or t^k we mean that t ranges over all types with that count on the topmost type constructor. Count annotated types are written using the following notation

$$t \ ::= \ Int^k \mid t_1 \times^k t_2 \mid \mu\alpha.^k c_1 \ t_1 + \cdots + c_n \ t_n \mid \alpha \mid t_1 \rightarrow^k t_2$$

where k is a count (0, 1 or ∞).

3.2 Annotated expressions

The aim of the analysis is to annotate value introducing expressions in the program by an usage count, that gives an upper limit to the number of uses of the value created by the expression. The syntax of annotated expressions is

$$\begin{array}{l} e ::= x \mid n^k \mid e_1 +^k e_2 \mid (e_1,^k e_2) \mid split \ e_1 \ to \ (x_1, x_2) \ in \ e_2 \\ \quad \mid \ c_i^k \ e \mid case \ e_0 \ of \ c_1 \ x_1 \ \rightarrow \ e_1; \ldots; c_n \ x_n \ \rightarrow \ e_n \\ \quad \mid \ e_1 \ e_2 \mid \lambda^k x : t.e \mid let \ x : t = e_1 \ in \ e_2 \\ \quad \mid \ letrec \ x_1 : t_1 = e_1, \ldots, x_n : t_n = e_n \ in \ e_0 \end{array}$$

where k is a count (0, 1 or ∞). Note that the types used in abstractions and let(rec) expressions are now annotated types.

The annotations on value-producing expressions (*i.e.* constants, additions, pairs, injected values and lambda-abstractions) indicate upper bounds of the usage of the values produced by these expressions. Annotated types on lambda-bound and let(rec)-bound variables indicate how many times values bound to these variables are used *inside their scope*. Since the same value may be bound to several variables, a single-use annotation on one variable does not guarantee that the value bound to it is single-threaded. Only the annotation at the creation point of a value will reflect the total usage of that value throughout its lifetime.

For some kinds of optimization it is sufficient to know the number of uses of a value at the creation point of that value, but for other optimizations it is required at a usage point to know if this is the single use. The latter information is not present in annotations as shown, but it is a simple matter to propagate this information to the uses of the value. In this respect (as well as in our use of a 0 count and *split*-expressions), our analysis is similar to the analysis by Jensen and Mogensen from ESOP'90 [3].

3.3 Counts

Apart from the obvious ordering ($0 \leq 1 \leq \infty$) we also need some operations on counts:

$k_1 \sqcup k_2$			
$k_1 \backslash k_2$	0	1	∞
0	0	1	∞
1	1	1	∞
∞	∞	∞	∞

$k_1 + k_2$			
$k_1 \backslash k_2$	0	1	∞
0	0	1	∞
1	1	∞	∞
∞	∞	∞	∞

$k_1 \cdot k_2$			
$k_1 \backslash k_2$	0	1	∞
0	0	0	0
1	0	1	∞
∞	0	∞	∞

$k_1 \rhd k_2$			
$k_1 \backslash k_2$	0	1	∞
0	0	0	0
1	0	1	∞
∞	0	1	∞

$\sqcup$ is least upper bound. $+$ and $\cdot$ are addition and multiplication over counts, respectively. $\rhd$ is used as a guard to indicate that uses in an expression should not count if the result of the expression is never used. Hence, we use this operator to simulate lazines of evaluation: An expression is not evaluated if its result is never used.

We extend the ordering and the operations to work on annotated types and type environments as well as counts. The ordering extends pointwise: For a type t_1 to be less than another t_2, the must have the same underlying type and the counts in t_1 must be less than the counts on the corresponding positions in t_2, with the exception of function types, where the ordering is contravariant, i.e.

$$(t_1 \rightarrow^{k_1} t_2) \leq (t_3 \rightarrow^{k_2} t_4) \quad \text{iff} \quad t_3 \leq t1,\ t2 \leq t4,\ k_1 \leq k_2$$

Ordering on type environments E_1 and E_2 implies the same underlying structure (*i.e.* the same variable is bound to types in E_1 and E_2 that are identical except for annotations) and $E_1 \leq E_2$ indicates that for all x, if E_1 binds x to t_1 and E_2 binds x to t_2 then $t_1 \leq t_2$.

Extending the operations on counts to work on types is complicated by the contravariance of the function types. Instead of defining exactly how the operations work on types, we will postpone this until we translate constraints on types

into constraints on counts in section 5. Constraints on environments expand directly into sets of constraints on the individual types in the obvious manner.

We use the following precedence order on the operators, from tightest to weakest binding: $\triangleright, \cdot, +, \sqcup, \leq$.

4 Type rules

We now present well-typedness rules for our language. The intention is that in any well-annotated program, the values created at an expression with annotation k will be used at most k times during evaluation under call-by-need. Unlike in [10], we allow the same variable to have different annotations on its type in different parts of the program. The annotations describe the usage in the given subexpression. It is hence the usage count in the type given at the creation time of a value that determines the total number of (potential) uses of the value.

The judgements are of the form

$$E \vdash e : t$$

where E is an environment mapping variables to annotated types, e is an annotated expression and t is an annotated type. The intuitive meaning is that if e is to be evaluated in a context given by t, then E describes the uses of the free variables of e.

If we assume that the result of the entire computation is printed, the natural context for the goal expression has usage 1 on all components of the result type.

In the rules below, we asume that the result of the entire expression is required. For non-strict uses of values, we use the $\triangleright$ guard on the type environments of subexpressions that may not always be evaluated. In this way, laziness is modeled. In the explanation about how the information is passed around, we state an intuitive order of information flow. The actual analysis will not use this order of evaluation but generate a set of constraints that are solved off-line. Hence, the flow-description is only intended to give an intuitive understanding of the type rules.

We will in general use inequality constraints between environments even in cases where equality would seem natural. The reason is that equality constraints are not in general efficiently solvable and a unique best solution may not exist. This point is explained in section 5.

The variable rule below simply indicates that if a variable is used in context t, then the environment reflects that use.

$$E[x : t] \vdash x : t$$

A constant expression indicates the number of uses of the constant.

$$E \vdash n^k : Int^k$$

If we add two numbers, these are used once each regardless of how many times we use the result.

$$\frac{E_1 \vdash e_1 \,:\, Int^1 \quad E_2 \vdash e_2 \,:\, Int^1 \quad E_1 + E_2 \leq E}{E \vdash e_1 +^k e_2 \,:\, Int^k}$$

The annotation on a pair expression must match the number of uses indicated by the type. The subexpressions are evaluated only if the corresponding parts of the pair are later accessed (indicated by a non-zero count on the types of the components). Hence, we guard the environments of the subexpressions by the counts on their types before adding them.

$$\frac{E_1 \vdash e_1 \,:\, t_1^{k_1} \quad E_2 \vdash e_2 \,:\, t_2^{k_2} \quad k_1 \rhd E_1 + k_2 \rhd E_2 \leq E}{E \vdash (e_1,^k e_2) \,:\, t_1^{k_1} \times^k t_2^{k_2}}$$

When analysing a *split* expression, we find the contexts of the components by analysing the body in the context of the entire expression. We then combine these to a product type with usage 1 (the usage implied by the *split*) and analyse the splitted expression. Since *split* is strict in both subexpressions, we use no guards.

$$\frac{E_1[x_1 : t_1, x_2 : t_2] \vdash e_2 : t \quad E_2 \vdash e_1 \,:\, t_1 \times^1 t_2 \quad E_1 + E_2 \leq E}{E \vdash split\ e_1\ to\ (x_1, x_2)\ in\ e_2 \,:\, t}$$

When injecting a value into a sum-type, we annotate the constructor by the usage count indicated by the type. Since injection is lazy, we guard the environment for the injected expression by its usage.

$$\frac{E_1 \vdash e \,:\, t_i \quad t_i \rhd E_1 \leq E}{E \vdash c_i^k\ e \,:\, \mu\alpha.^k c_1\ t_1 + \cdots + c_i\ t_i + \cdots + c_n\ t_n}$$

When analysing a *case*-expression, we analyse each of the branches using the context of the entire expression. The yielded contexts for each of the pattern variables are used to construct a recursive type for the sum-type, constraining each of the injected types to be conservative approximations of the pattern-variable contexts and adding the constraint that a value of that type may be used at least once (namely, by this *case*-expression). This type is used as the context for the inspected expression. The environment for the entire expression is found as the combined worst-case of the environments of the branches plus the environmnet of the inspected expression. The i subscript ranges from 1 to n.

$$\frac{\begin{array}{c} E_i[x_i : s_i] \vdash e_i \,:\, t \quad s_i \leq t_i[\alpha \backslash (\mu\alpha.^k c_1\ t_1 + \cdots + c_n\ t_n)] \quad 1 \leq k \\ E_0 \vdash e_0 \,:\, \mu\alpha.^k c_1\ t_1 + \cdots + c_n\ t_n \quad E_0 + E_i \leq E \end{array}}{E \vdash case\ e_0\ of\ c_1\ x_1\ \rightarrow\ e_1; \ldots; c_n\ x_n\ \rightarrow\ e_n \,:\, t}$$

When analysing a function application, we find the type of the function to get a context for the argument. We also constrain the result type of the function to be at least as "bad" as the context of the application. Since function application is lazy, we guard the environment of the argument expression.

$$\frac{E_1 \vdash e_1 \,:\, t_1^k \to^1 t_2 \quad E_2 \vdash e_2 \,:\, t_1^k \quad t_3 \leq t_2 \quad E_1 + k \rhd E_2 \leq E}{E \vdash e_1\ e_2 \,:\, t_3}$$

An abstraction is annotated both with the number of uses of the function and by the type of the argument. The body is analysed to find the uses of variables in a single evaluation of the body. Since the body is evaluated at each application of the function, we multiply the environment for the body by the number of uses of the function.

$$\frac{E_1[x : t_1] \vdash e \,:\, t_2 \quad k \cdot E_1 \leq E}{E \vdash \lambda^k x : t_1.e \,:\, t_1 \to^k t_2}$$

We analyse the body of a *let*-expression to find the context for the bound variable. This is used as the context for the bound expression and to annotate the variable in the expression. Since *let*-exressions are lazy, we guard the environment of the bound expression by its type.

$$\frac{E_2[x : t_1^k] \vdash e_2 : t_2 \quad E_1 \vdash e \,:\, t_1^k \quad k \rhd E_1 + E_2 \leq E}{E \vdash let\ x : t_1^k = e_1\ in\ e_2 \,:\, t_2}$$

We assume that strongly connected components of a *letrec* are found in an earlier stage of compilation and used to split complex *letrec*s into nested *let*s and *letrec*s, as described in [4]. This allows us to use a fairly simple conservative approximation and assume that the variables bound in a *letrec*-expression can be used arbitrarily often. Hence, we constrain the types of the bound expressions to have infinite counts if they are used at all. The subscript i ranges from 0 to n and j ranges from 1 to n.

$$\frac{E_i[x_1 : t_1, \ldots, x_n : t_n] \vdash e_i \,:\, s_i \quad \infty \cdot t_j \leq s_j \quad E_0 + \cdots + E_n \leq E}{E \vdash letrec\ x_1 : t_1 = e_1, \ldots, x_n : t_n = e_n\ in\ e_0 \,:\, s_0}$$

5 Usage analysis

The type rules can be used to check whether a given annotation is valid, but it is more interesting to use the rules to find a minimal valid annotation of a program.

The approach we have chosen is to write a type derivation of the program where all counts are replaced by different variables. This is possible, as type derivations for different annotations of the same program only differ in the values of the counts used in types, environments and annotations. The type rules can then be seen as generating a set of constraints between counts, types and environments, which we have to solve.

We will expand constraints over environments and types to collections of constraints over counts. Constraints on environments are expanded pointwisely to constraints over the individual types (when one of the arguments to an operator

is an atomic count, this is distributed over all the types in the environment). Constraints on types are likewise expanded pointwisely to component types (i.e. components of product and sum types), with the exception of function types where the contravariancy of the ordering complicates matters. Also, some operations don't naturally expand pointwisely on function types.

If we, for example, add two function types using the + operator, we don't just add the components. The reason is that the counts on the argument and result types of a function type indicate how many times the argument and result of the function will be used when the function is called. But adding two function types just means that the function is used twice, which doesn't mean that its argument will be used any more often, as different arguments may be used in different calls. Hence, we must just use the worst case and hence take the least upper bound of the result counts and (by contravariance) the greatest lower bound of the argument types. When we combine the addition with a $\leq$ to get a constraint, we get that:

$$(t_1 \to^{k_1} t_2) + (t_3 \to^{k_2} t_4) \leq (t_5 \to^{k_3} t_6)$$

is expanded by a combination of the contravariant comparison rule and the expansion rule for + into

$$k_1 + k_2 \leq k_3,\ t_5 \leq t_1 \sqcap t_3,\ t_2 \sqcup t_4 \leq t_6$$

Which we by using the relation between $\sqcap$ and $\leq$ expand further to

$$k_1 + k_2 \leq k_3,\ t_5 \leq t_1,\ t_5 \leq t_3,\ t_2 \sqcup t_4 \leq t_6$$

When expanding the constraint

$$k_0 \rhd (t_1 \to^{k_1} t_2) \leq t_3 \to^{k_2} t_4$$

we use the intuition that if $k_0 = 0$ then the constraint should be satisfied by all types and if $k_0 > 0$, then the constraint reduces to a simple $\leq$ constraint. Hence, we expand the above constraint to

$$k_0 \rhd k_1 \leq k_2,\ k_0 \rhd t_3 \leq t_1,\ k_0 \rhd t_2 \leq t_4$$

Note how the contravariancy is handled by moving the $\rhd$ operator to t_3.

Similarly, we use the intuition that $0 \cdot t_1 \leq t_2$ should be trivially satisfied, $1 \cdot t_1 \leq t_2$ is equivalent to $t_1 \leq t_2$ and $2 \cdot t_1 \leq t_2$ is equivalent to $t_1 + t_1 \leq t_2$ to expand

$$k_0 \cdot (t_1 \to^{k_1} t_2) \leq t_3 \to^{k_2} t_4$$

into

$$k_0 \cdot k_1 \leq k_2,\ k_0 \rhd t_3 \leq t_1,\ k_0 \rhd t_2 \leq t_4$$

When we have constraints that have more than one operator on the left hand side of the inequality sign (e.g. $k \rhd E_1 + E_2 \leq E$ or n-ary sums), we use the monotinicity of the operators to expand this into a series of simpler constraints by inserting new variables. For example, $k \rhd E_1 + E_2 \leq E$ is expanded to $k \rhd E_1 \leq E'$, $E' + E_2 \leq E$, where E' is a new environment which has the same structure as E_1, E_2 and E_3.

Each constraint is now in one of the forms $k_1 = k_2$, $1 \leq k_1$, $k_1 \leq k_2$, $k_1 + k_2 \leq k_3$, $\infty \cdot k_1 \leq k_2$, $k_1 \cdot k_2 \leq k_3$, $k_1 \sqcup k_2 \leq k_3$ or $k_1 \rhd k_2 \leq k_3$, where the k_i are variables that range over counts. Each solution to a set of constraints generated by the type rules corresponds to a type derivation for a well-annotated program.

We want minimal annotations, so we must find a minimal solution to the constraint set. We have carefully chosen the constraints to be meet-closed: Whenever we have two solutions, then the meet (greatest lower bound) of these is also a solution. This is the motivation for using $k_1 + k_2 \leq k_3$ where $k_1 + k_2 = k_3$ might seem more natural *etc.*, as the latter is not meet-closed. Hence, a unique minimal solution can be found by taking the meet of all solutions. That a solution must exist is not terribly difficult to see either: All occurences of constants in the constraints will occur on the left of an inequality, so setting all count variables to ∞ will yield a solution.

Each type rule generates a constant number of constraints between types and type environments. As we have chosen a language where varibles are typed explicitly in the program text, the size (measured as number of nodes in the syntax tree) of a type is bounded by the size of the program. Furthermore, the sum of the sizes of types in a type environment is also bounded by the size of the program, as the type environment lists a subset of the variables from the program with their types. Hence, each constraint between types or environments expands to a number of primitive constraints that is bounded linearly by the size of the program. Since the number of inferences in a type derivation is proportional to the size of the program, the total number of primitive constraints is bounded by the square of the size of the typed program. As we will see that the constraints can be solved in time linear in the size of the constraint set, the analysis can be done in quadratic time.

5.1 Solving the constraints

Rehof and Mogensen [7] show two different ways of solving meet-closed inequality constraints over finite lattices. One method solves the constraints directly, using a variant of Kildall's algorithm [5]. The other translates the constraints into Horn-clauses, which can be solved using the HORNSAT algorithm of Dowling and Gallier [2]. Both methods find the least solution in time linear in the size of the constraint set. We will show the latter method here and refer readers to [7] for the first.

We represent each element in the count lattice by two boolean values:

count	representation
0	00
1	01
∞	11

Hence, we replace each constraint variable k_i by two variables l_i and r_i over the binary domain, with the constraint $l_i \leq r_i$ to reflect that the pair 10 is unused.

We translate each constraint over counts to a set of constraints over the boolean lattice. The translation below uses the the general scheme shown in [7] with local reduction in the cases where constants occur in the constraints:

constraint	translation
$k_1 = k_2$	$l_1 \leq l_2,\ r_1 \leq r_2,\ l_2 \leq l_1,\ r_2 \leq r_1$
$1 \leq k$	$1 \leq r$
$k_1 \leq k_2$	$l_1 \leq l_2,\ r_1 \leq r_2$
$k_1 + k_2 \leq k_3$	$l_1 \leq l_3,\ r_1 \leq r_3,\ l_2 \leq l_3,\ r_2 \leq r_3,\ r_1 \wedge r_2 \leq l_3$
$k_1 \sqcup k_2 \leq k_3$	$l_1 \leq l_3,\ r_1 \leq r_3,\ l_2 \leq l_3,\ r_2 \leq r_3$
$\infty \cdot k_1 \leq k_2$	$r_1 \leq l_2$
$k_1 \cdot k_2 \leq k_3$	$r_1 \wedge r_2 \leq r_3,\ l_1 \wedge r_2 \leq l_3,\ l_2 \wedge r_1 \leq l_3$
$k_1 \rhd k_2 \leq k_3$	$r_1 \wedge l_2 \leq l_3,\ r_1 \wedge r_2 \leq r_3$

Note that we have reduced the number of different constraints to just ordering ($\leq$ is the usual ordering on the binary domain) and $x \wedge y \leq z$, where $x \wedge y$ is 0 if any of x or y is 0, and 1 otherwise.

It is shown in [7] that, given the extra constraints $l_i \leq r_i$ for all variables k_i, the translation has a solution if and only if the original constraint has. Furthermore, any solution to the translation maps into a solution of the original constraint set using the inverse of the representation function.

6 Conclusion

We can solve these constraints in linear time using a variety of methods. The number of constraints is linear in the size of the type derivation for the program we analyse, i.e. in the worst case quadratic in the size of the typed program. This is somewhat worse than Turner em et al.'s analysis [10] which is linear in the size of the typed program. Our analysis is, however, more precise: Whenever Turner *et al.*'s analysis detects single use, so will our, and there are cases (e.g. the *mean* example) where our analysis detects single use where Turner *et al.*'s analysis do not.

The presented version of the analysis finds usage counts at value creation times. This is fine if the update information is put into the values themselves in the form of self-updating closures, as is the case for most tagless implementations of lazy functional languages. If usage counts are required at value destruction time, the presented analysis must be supplemented by a simple forwards analysis. This approach was also used in [3]. Note that this will actually reduce the

precision of the analysis, as values with different usages may float to the same destruction point. If annotations are given at creation time, the destruction can treat these differently, but any annotation at the destruction point must treat the values the same way.

The analysis can detect that a partial application is single-use. In addition to optimizing call-by-need to call-by-name, that information can be used to avoid full-laziness transformation of the function, which can save both space and time.

Initial experiments with an implementation of the constraint solver from [7] indicate that solving the constraints directly takes approximately the same time as translating the constraints to Horn-clauses and then using the first of the linear-time HORNSAT algorithms from [2]. However, the latter method uses somewhat more space. These experiments use randomly generated constraints, not constraints generated by an actual usage count analysis, so it is too early to tell which will be best for an actual implementation.

Even though the worst case analysis time is quadratic, we believe this is rarely encountered in real programs, as types and environments typically grow less than linearly in program size. There is, however, little doubt that the analysis of [10] is faster, and it is not clear to what extent the added precision of the proposed analysis is worth the extra cost.

In addition to using the analysis to avoid updates during graph-reduction, it can be used for a limited kind of compile-time garbage collection. This was, indeed, the stated intention of the analysis presented in [3]. In a system with a good garbage collection, this kind of compile-time GC is probably not worth the effort. However, if the typing rules are modified to handle a call-by-value language (essentially eliminating the uses of $\triangleright$), the analysis can be used in conjunction with region inference [9]. Experience with the ML-kit with regions [8] shows that a common space leak is when a function takes apart a constructor and builds a new constructor of the same type in the same region, causing this region to grow. If the analysis can detect that the original constructor is not used again, the same space can be reused for the new constructor, avoiding the growth.

Another related type system is the uniqueness typing of Clean [1]. This too has the restriction that if the spine of a list is used repeatedly, then it is assumed that this is also the case for the elements. However, the uniqueness type analysis takes evaluation order into account and can hence in some cases detect the last use of value even if there are multiple uses within the life time of the value, which our analysis can not. Furthermore, the uniqueness type system can handle polymorphism, which the analysis presented here does not.

A possible way of handling polymorphism is by abstracting over annotated types and include in each type schema a set of constraints over the type variables (both bound and free). These constraints will be added to the current set of constraints when a type schema is instantiated. When type variables are instantiated to actual types, constraints over the type variables are expanded out into more primitive constraints. In the end (assuming the goal function of the program has a monomorphic type), all constraints are expanded fully into constraints over atomic counts which can be solved as described above. This

approach is likely to have a high worst-case complexity, but may be O.K in practice. It may be worthwhile to reduce the constraints in type schemas at the time they are constructed rather than wait until they are fully instantiated. This will, however, require a different solution mechanism, as the current constraint solver is inherently off-line.

References

1. Erik Barendsen and Sjaak Smetsers. Uniqueness type inference. In *PLILP'95, Utrecht, The Netherlands (Lecture Notes in Computer Science, vol. 982)*, pages 189–206. Springer-Verlag LNCS 982, 1995.
2. W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, (3):267–284, 1984.
3. Thomas P. Jensen and Torben Æ. Mogensen. A backwards analysis for compile-time garbage collection. In *ESOP '90, Copenhagen, Denmark (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag LNCS 432, 1990.
4. Simon L. Peyton Jones and David Lester. *Implementing Functional Languages.* Prentice Hall Series in Computer Science. Prentice Hall, New York, London, Toronto, Syney, Tokyo, Signapore, 1 edition, 1992.
5. G. Kildall. A unified approach to global program optimization. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, January 1993.
6. M.J. Plasmeijer and M.C.J.D. van Eekelen. Concurrent clean 1.0 language report. Technical report, Computing Science Institute, University of Nijmegen, 1995.
7. Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semi-lattices. In R. Cousot and D.A. Sscmidt, editors, *Third International Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 285–301. Springer, September 1996.
8. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical report, Dept. of Computer Science, University of Copenhagen, 1997.
9. Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, January 1994.
10. David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.