Analysis of Recursive Types in Lisp-like Languages

Edward Wang *Paul N. Hilfinger*

Computer Science Division
University of California
Berkeley, CA 94720
edward@CS.Berkeley.EDU
hilfingr@CS.Berkeley.EDU

Abstract

We introduce a new algorithm to analyze recursive, structured types. It derives information from object uses (accesser functions with type checking), as well as from object allocation. The type description is a form of graph grammar and is naturally finite even in the presence of loops. The intended use of the algorithm is to discover and remove unnecessary type checks, but it can be augmented to provide alias information as well.

1 Introduction

We are interested in the behavior of programs in an untyped language with data structures constructed from records containing pointers, typified by Lisp programs that manipulate structures built from cons cells. In this paper, we will introduce a new algorithm to analyze such programs. Unlike many type-inference algorithms, it is designed to work on partial programs (with or without declarations) and to produce partial inferences. The main goal is to remove unnecessary type checks.

The algorithm uses information from both object creation and the implicit type checks in object uses, and naturally produces a bounded description (bounded by a function on program size) even for arbitrarily large structures. As a result, it can gather more precise information than other algorithms. For example, it is capable of deducing a description of a list from a loop over that list, and using this information to remove type checks from subsequent loops over the same list.

To emphasize an intuitive understanding of the algorithm, we begin this paper with a series of examples (Section 2), and a discussion of related algorithms and theories (Section 3). The rest of the paper, beginning with Section 4, deals with the formalism that is the foundation of the algorithm and shows some of its consequences. Readers who wish only a general understanding of our work will find Section 2 sufficient and painless.

2 Some Examples

These initial examples are in a form of Lisp assembly language with assignment statements, labels, and conditional and unconditional gotos. For now, we only consider three types: cons, nil, and other. In Lisp, `car` and `cdr` are overloaded on cons and nil. To avoid this complication, we will use selector functions (called **head** and **tail**) defined only on cons cells. Also, we will make all type checks explicit. A type error is indicated by a branch to an error routine that does not return.

Our type description is a form of grammar. At each program point, the algorithm computes a set of productions (called a *p-set*). In these productions, the terminals are the type symbols (`cons`, `nil`, `other`). The nonterminals are *sets* of variables and definitions (a definition being a symbol that represents a particular assignment to a variable).[1] We write a definition with a numeric superscript over the variable symbol (like $\mathbf{x}^1$). The empty set, $\emptyset$, is a legal nonterminal. As we will see, it naturally represents an unknown value. All p-sets contain the productions

$$\begin{aligned}\emptyset &\rightarrow \texttt{cons}\ \emptyset\ \emptyset\\ \emptyset &\rightarrow \texttt{nil}\\ \emptyset &\rightarrow \texttt{other}\end{aligned}$$

which we can also write more concisely:

$$\emptyset \rightarrow \texttt{cons}\ \emptyset\ \emptyset \mid \texttt{nil} \mid \texttt{other}$$

We also require the right-hand side of a production to be a single terminal followed by a sequence of nonterminals (corresponding to the slots contained in the type).

[1] Each assignment to a variable is a different definition of that variable. For example, a variable **x** has two definitions if it appears on the left-hand sides of two assignment statements.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0216...$1.50

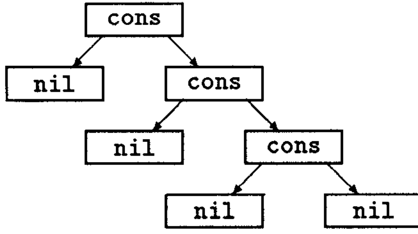

Figure 1: A list of three nils.

A variable x with an unknown value can be described like this:

$$\{x\} \rightarrow \text{cons } \emptyset\ \emptyset \mid \text{nil} \mid \text{other}$$
$$\emptyset \rightarrow \text{cons } \emptyset\ \emptyset \mid \text{nil} \mid \text{other}$$

Without being too precise, we can see that every possible value in our domain corresponds to some derivation of $\{x\}$. For example, let x be a list of three nils (Figure 1). The preorder traversal of the list (as a tree) is the string

cons nil cons nil cons nil nil,

which is a derivation of $\{x\}$.

To illustrate the interaction between variables, we use a p-set from a later example (Figure 2):

$$\begin{aligned}
&\{x, y, x^1, y^1\} \rightarrow \text{nil}\\
&\{y, x^1, y^1\} \rightarrow \text{cons } \emptyset\ \{x^2\}\\
&\{y, x^1, y^1\} \rightarrow \text{cons } \emptyset\ \{x, x^2\}\\
&\{x^2\} \rightarrow \text{cons } \emptyset\ \{x^2\}\\
&\{x^2\} \rightarrow \text{cons } \emptyset\ \{x, x^2\}\\
&\{x, x^2\} \rightarrow \text{nil}\\
&\emptyset \rightarrow \text{cons } \emptyset\ \emptyset \mid \text{nil} \mid \text{other}
\end{aligned}$$

Let's call the first six productions p_1 to p_6. There are two variables, x and y. Their possible values are describe by the same p-set. The nonterminals describing a variable are those that contain that variable. The nonterminals for y are $\{x, y, x^1, y^1\}$ and $\{y, x^1, y^1\}$. The first also describes x. Therefore, by expanding p_1, we see that one possibility is that both x and y are nil. We get another possible value of y by expanding p_3 then p_6:

$$\begin{aligned}
\{y, x^1, y^1\} &\Rightarrow \text{cons } \emptyset\ \{x, x^2\}\\
&\Rightarrow \text{cons } \emptyset \text{ nil}
\end{aligned}$$

So y is a list of length one, and x is the terminating nil (because it comes from $\{x, x^2\}$). We get yet a third value by expanding p_2, p_5, then p_6:

$$\begin{aligned}
\{y, x^1, y^1\} &\Rightarrow \text{cons } \emptyset\ \{x^2\}\\
&\Rightarrow \text{cons } \emptyset \text{ cons } \emptyset\ \{x, x^2\}\\
&\Rightarrow \text{cons } \emptyset \text{ cons } \emptyset \text{ nil}
\end{aligned}$$

Here, y is a list of length two, and x is again the final nil (because the only nonterminal containing x expands to it). In general, for each variable, a derivation of a p-set is only allowed to use a single nonterminal containing that variable. Continuing the process, we can see that y is always a list, and x is always the terminating nil.

In the rest of this section, we will omit the three productions for $\emptyset$ from the p-sets, since they are always the same and always present.

Because the specific definitions of variables are important to us, all examples will be annotated with definition indices. As we will see, most of the algorithm involves the construction of nonterminals out of these definitions and variables.

We can now analyze this piece of code:

```
if not consp(x) goto error
y¹ ← tail(x)
```

Two things happen here. First, after the type check, we know that x must be a cons cell. Second, after the assignment we know that y and the tail of x point to the same object.

The algorithm annotates the program points. We represent this with a p-set between each pair of sequential statements, and two p-sets after each conditional (one for the true branch and one for the false branch). This is the result:

```
    {{x,x¹} → cons ∅ ∅ | nil | other}
if not consp(x) goto error
    {{x,x¹} → nil | other}
    {{x,x¹} → cons ∅ ∅}
y¹ ← tail(x)
    {{x,x¹} → cons ∅ {y,y¹}
     {y,y¹} → cons ∅ ∅ | nil | other}
```

Each p-set describes the possible program states at *that* point in the program. They are not universal assertions (like a variable declaration) or expectations that must hold to prevent future errors. As before, every possible value of a variable x is a derivation of some nonterminal containing x.

In the example, the initial condition shows an unknown x, from (an unseen) definition x^1. The algorithm is a forward flow analysis. The conditional filters out the productions for x that fail the type check. The assignment ($y^1 \leftarrow$ tail(x)) makes y equivalent to the tail of x. The nonterminal $\{y, y^1\}$ is synthesized and takes the place of $\emptyset$ in the tail part of ($x \rightarrow$ cons $\emptyset\ \emptyset$), and the three (omitted) productions for $\emptyset$ are copied to make the productions for $\{y, y^1\}$.

A feature of the type description is that all variables are described by the same p-set. This is indeed necessary if structural information is to be inferred. In the above example, any subsequent restriction on the possible types of y will also add to the information on x:

```
    {{x,x^1} → cons ∅ {y,y^1}
     {y,y^1} → cons ∅ ∅ | nil | other}
if not consp(y) goto error
    {{x,x^1} → cons ∅ {y,y^1}
     {y,y^1} → nil | other}
    {{x,x^1} → cons ∅ {y,y^1}
     {y,y^1} → cons ∅ ∅}
y^2 ← tail(y)
    {{x,x^1} → cons ∅ {y^1}
     {y^1} → cons ∅ {y,y^2}
     {y,y^2} → cons ∅ ∅ | nil | other}
```

Now we know that **x** must be a list of at least two elements—specifically, **x** is a cons cell, the tail of which is an anonymous cons cell, and the tail of that is an unknown object pointed to by **y**. The nonterminal $\{\mathtt{y}, \mathtt{y}^1\}$ loses the **y** after the second assignment, because it no longer describes the *current* value of **y**.

The general procedure for processing assignment statements is defined in Section 6. We give a simplified version here. Let x and y be distinct variables, and P the in-coming p-set. For the simple assignment ($x^d \leftarrow y$), the algorithm first deletes x from all nonterminals in P, then adds $\{x, x^d\}$ to all nonterminals containing y. For the statement ($x^d \leftarrow \mathtt{tail}(y)$), suppose y is described by a single production $p \in P$ ($p = (u \rightarrow \mathbf{cons}\ v\ w)$ and $y \in u$). Again, we delete x from all nonterminals. The rest depends on w:

- If w contains a variable, then replace all occurrences of w by $w \cup \{x, x^d\}$ in all productions. For example,

```
        {{x,x^1} → other
         {y,y^1} → cons ∅ {z,z^1}
         {z,z^1} → nil}
    x^2 ← tail(y)
        {{x^1} → other
         {y,y^1} → cons ∅ {x,z,x^2,z^1}
         {x,z,x^2,z^1} → nil}.
```

- If w does not contain a variable, then replace w by $w \cup \{x, x^d\}$ in p, copy all productions with w on the left-hand side, and replace w by $w \cup \{x, x^d\}$ on the left-hand sides of the copies. For example,

```
        {{x,x^1} → other
         {y,y^1} → cons ∅ {z^1}
         {z^1} → nil}
    x^2 ← tail(y)
        {{x^1} → other
         {y,y^1} → cons ∅ {x,x^2,z^1}
         {x,x^2,z^1} → nil
         {z^1} → nil}.
```

  The case of $w = \emptyset$ is just one instance of this general rule.

The reason for having two cases is that a nonterminal with a variable is, roughly speaking, more specific.

```
    {{y,y^1} → cons ∅ ∅ | nil | other}
x^1 ← y
    {{x,y,x^1,y^1} → cons ∅ ∅ | nil | other}
L1:
    {{x,y,x^1,y^1} → cons ∅ ∅ | nil | other
     {y,x^1,y^1} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x^2} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x,x^2} → cons ∅ ∅ | nil | other}
if null(x) goto L2
    {...}
    {{x,y,x^1,y^1} → cons ∅ ∅ | other
     {y,x^1,y^1} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x^2} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x,x^2} → cons ∅ ∅ | other}
if not consp(x) goto error
    {...}
    {{x,y,x^1,y^1} → cons ∅ ∅
     {y,x^1,y^1} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x^2} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x,x^2} → cons ∅ ∅}
x^2 ← tail(x)
    {{y,x^1,y^1} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x^2} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x,x^2} → cons ∅ ∅ | nil | other}
goto L1
L2:
    {{x,y,x^1,y^1} → nil
     {y,x^1,y^1} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x^2} → cons ∅ {x^2} | cons ∅ {x,x^2}
     {x,x^2} → nil}
```

Figure 2: The loop after analysis.

Variable-free nonterminals are allowed to generate more than one object. Ones with variables must generate a single object, or none at all. The algorithm will still be correct if it always follows the second case, but it will be less precise.

The next example is a loop, for which a compact description of an arbitrarily large structure will be generated:

```
x^1 ← y
L1:
if null(x) goto L2
if not consp(x) goto error
x^2 ← tail(x)
goto L1
L2:
```

In the loop, the index variable **x** is used to iterate down the list **y**. We will analyze two such loops in sequence, to show the p-sets generated and how they are used to remove an unnecessary type check. The results are shown in Figures 2 and 3. Because of the length of this example, we have omitted some of the p-sets (those

```
    {{x,y,x¹,y¹} → nil
     {y,x¹,y¹} → cons ∅ {x²} | cons ∅ {x,x²}
     {x²} → cons ∅ {x²} | cons ∅ {x,x²}
     {x,x²} → nil}
x³ ← y
    {{x,y,x¹,x³,y¹} → cons ∅ {x²} | nil
     {x²} → cons ∅ {x²} | nil}
L3:
    {...}
if null(x) goto L4
    {...}
    {{x,y,x¹,x³,y¹} → cons ∅ {x²}
     {x²,x⁴} → cons ∅ {x²,x⁴} |
               cons ∅ {x,x²,x⁴}
     {x,x²,x⁴} → cons ∅ {x²}
     {x²} → cons ∅ {x²} | nil}
if not consp(x) goto error
    ∅
    {...}
x⁴ ← tail(x)
    {{y,x¹,x³,y¹} → cons ∅ {x²,x⁴} |
                    cons ∅ {x,x²,x⁴}
     {x²,x⁴} → cons ∅ {x²,x⁴} |
               cons ∅ {x,x²,x⁴}
     {x,x²,x⁴} → cons ∅ {x²} | nil
     {x²} → cons ∅ {x²} | nil}
goto L3
L4:
    {{x,y,x¹,x³,y¹} → nil
     {y,x¹,x³,y¹} → cons ∅ {x²,x⁴} |
                    cons ∅ {x,x²,x⁴}
     {x²,x⁴} → cons ∅ {x²,x⁴} |
               cons ∅ {x,x²,x⁴}
     {x,x²,x⁴} → nil}
```

Figure 3: The second loop. The initial condition is the final condition of the first loop (Figure 2).

```
x¹ ← y
L1:
if null(x) goto L2
if not consp(x) goto error
x² ← tail(x)
if not consp(x) goto error
x³ ← tail(x)
goto L1
L2:
    {{x,y,x¹,y¹} → nil
     {y,x¹,y¹} → cons ∅ {x²}
     {x²} → cons ∅ {x³} | cons ∅ {x,x³}
     {x³} → cons ∅ {x²}
     {x,x³} → nil}
z¹ ← y
L3:
if null(z) goto L4
if not consp(z) goto error
z² ← tail(z)
if not consp(z) goto error
z³ ← tail(z)
if not consp(z) goto error
z⁴ ← tail(z)
goto L3
L4:
    {{x,y,z,x¹,y¹,z¹} → nil
     {y,x¹,y¹,z¹} → cons ∅ {x²,z²}
     {x²,z²} → cons ∅ {x³,z³}
     {x³,z³} → cons ∅ {x²,z⁴}
     {x²,z⁴} → cons ∅ {x³,z²}
     {x³,z²} → cons ∅ {x²,z³}
     {x²,z³} → cons ∅ {x³,z⁴} |
               cons ∅ {x,z,x³,z⁴}
     {x³,z⁴} → cons ∅ {x²,z²}
     {x,z,x³,z⁴} → nil}
```

Figure 4: The *lcm*(2,3) example. Not all p-sets are shown.

shown as {...}). At a few places, we've also removed unused productions (those with variable-free left-hand sides not referred to in any right-hand sides).

Since we have already used the final p-set of the first loop in earlier discussions, we know that it correctly shows that **y** is a list and **x** points to the final nil. The end of the second loop (label L4 in Figure 3) shows the same thing but with different nonterminals.

A simple trick makes the p-sets immediately usable for optimization. When no production satisfies a conditional (for example, the type check in Figure 3), we simply make the entire p-set for that branch empty. This serves as a marker for dead code and also prevents information from (incorrectly) propagating further. Thus, edges that can never be traversed at run time are marked with empty sets. At the end of the analysis, it is a simple matter to delete dead statments and remove conditionals with only one branch.

One last example illustrates the power of this method. The first part of Figure 4 (up to label L2) is a loop that fails unless **y** is of even length. If this is followed by a loop that forces **y** to be a multiple of (in this case) three in length, then the combined result should be a list of some multiple of six. As shown in Figure 4, the algorithm is indeed capable of deducing this.

3 Background

Some recent authors on type analysis for Lisp have divided type inference algorithms into two classes: func-

tional and imperative [2, 11].[2] The functional methods are typified by Milner's type-inference algorithm (and its variants) for ML [15]. They are for functional languages (the only mutable objects in ML are references, and they are not truly polymorphic), and they are designed to find a single type (though possibly with free type variables) for every expression in the program. Type-correct Lisp programs are not always typable by such algorithms. However, it is not impossible to adapt ML type inference to an imperative language and to find partial solutions. For example, Johnson's *type flow* analysis includes an application of a unification algorithm to Lisp [5], and Gomard has developed an algorithm specifically for partial inference of otherwise untypable functional programs [4]. Related to our work, there is also the algorithm of Mishra and Reddy, which can deal with recursive structures [16].

In this classification, the imperative methods are based on algorithms of Kaplan and Ullman [8, 9] and of Miller [14]. In Common Lisp terms, they deal mainly with generic functions (like arithmetic operators that are overloaded on many types). As such, they must start with values of known types (literals or user-supplied declarations), then infer the types of other values. The goal is to reduce the cost of generic-function dispatch, and to do it with only a few user-supplied declarations. When applied to Lisp, these algorithms also tend to restrict the programming style by imposing a type discipline.

Falling outside this taxonomy, our algorithm concentrates on structures built from record-like objects. In Lisp, functions dealing with these types are typically monomorphic or weakly polymorphic. The implicit type operations are type checks rather than dispatches. They succeed if and only if they are given objects of the right types. Thus it is possible to deduce useful type information without any declarations. The optimization removes the unnecessary type checks. We seek not to change the Lisp programming style, but rather to support it, by making Lisp programs more efficient.

The type information for this style of analysis comes from two sources: object creation and object use. Algorithms that use only creation information keep track of structures as they are built (as in Jones and Muchnick [7]). The result can then be used to remove type checks when the same structures are used. This works well if the entire program can be analyzed completely, and if all data structures are created using allocation functions within the program. Usage information for type analysis is first noted by Kaplan and Ullman (as *side-way* inference). It can locally augment a global analysis. It also allows better results when fragments of a program are analyzed in isolation, which is almost a necessity with Lisp. Our algorithm uses both kinds of information.

The origin of type analysis from usage can be traced to constant propagation. Indeed, the algorithm of Wegman and Zadeck [17] can be used directly for type analysis.[3] However, this gives only the surface types of variables, not entire structures. To analyze whole structures, an algorithm must keep track of the relationship between variables. As we have seen in Section 2, the deep structure of the value of one variable is inferred from operations on other variables. A constant propagation algorithm that treats variables independently (e.g., using a constant-cross-variable product lattice) cannot do this.

In addition to propagating type assertions forward, some algorithms also propagate type expectations backward [6, 8, 9, 14]. This allows two useful optimizations: moving several identical type checks to a common point, and hoisting loop-invariant type checks out of the loop. For us, however, the advantage is small. Well-known optimization techniques, like common subexpression analysis and hoisting of loop-invariant computations, are specifically designed for these tasks. In addition, if the type checks for a list are moved out of the loop accessing it, a new loop just for the checks will then be needed. It is much better to share the loop overhead by leaving them where they are.

The presence of usage information makes an algorithm qualitatively different. The propagation of usage information necessarily follows *control flow*, because in some sense the information derived from type checks is a side effect of those checks. On the other hand, information from object creation, declarations, and literals follows the order of *data flow*: from definition to use, and from function arguments to function result.

Type analysis, alias analysis, and lifetime analysis algorithms all employ structural description of some form. Directed graphs [3, 7, 10], tree grammars [7], and string grammars [10] have all been used. Whatever the purpose, a common feature is that they all represent possibly unbounded structures with a bounded description. The more elegant of the algorithms derive compact descriptions naturally, without a forced cut-off at some predetermined size. Our description is a grammar of directed graphs, with some *ad hoc* features. It is naturally bounded.

To our knowledge, the only practical optimizing Lisp compiler that uses real type analysis is the Python compiler of the CMU Common Lisp [13]. It uses an algorithm based on constant propagation and imperative type inference [12].

To summarize, our algorithm is new in that it derives a structural description from both object creation and

[2] And for their work, the same authors have chosen to use imperative algorithms.

[3] Our trick of marking unexecuted edges with empty p-sets is a variant of the *execute* flag in their algorithm.

object use. The algorithm is a traditional flow analysis, and makes heavy use of the imperative aspects of Lisp. Its goal is to remove unnecessary type checks.

4 General Notation

A function is a set of ordered pairs. The image of function f from a subdomain A is written $f[A]$, and is equal to $\{f(a) : a \in A\}$. We also mean by $f\binom{a}{b}$ the function f with the value at a replaced by b:

$$f\binom{a}{b} = (f - \{(a, f(a))\}) \cup \{(a, b)\}.$$

Note that f need not be already defined at a.

The power set of a set A is written $\mathcal{P}(A)$.

5 Formal Semantics

We begin by defining the meaning of variable and storage. We assume a set of types, T. An object of type $t \in T$ is a record of k slots (the *arity* of t, which may be zero).

Definition 1 *A* storage graph *is a directed graph G with node labels and ordered out-edges. Specifically, G is a function on a finite set of vertices (called $|G|$) to a set of tuples, such that if $G(n) = (t, n_1, \ldots, n_k)$, then t is a k-ary type in T, and $n_i \in |G|$ for all $1 \leq i \leq k$. Also, we define $type^G(n) = t$, and $dest_i^G(n) = n_i$ for all $1 \leq i \leq k$.*

When the graph G is understood, we will often omit it and simply write $type(n)$ and $dest_i(n)$. To avoid set-theoretic difficulties later, we need to restrict the universe of vertices somewhat. Specifically, we say there is a countably infinite set V, such that for any storage graph G, $|G| \subset V$. Also, let the elements of V be ordered.

Definition 2 *Let X be the set of variables in a given program. A* storage state *for the program is a pair (G, f), where G is a storage graph and f is a function on X to $|G|$.*

We can now recast an earlier example in the new machinery. The list of three nils in Figure 1 can be redefined as a storage graph G (see also Figure 5):

$$\begin{aligned}
G(n_1) &= (\texttt{cons}, n_5, n_2),\\
G(n_2) &= (\texttt{cons}, n_6, n_3),\\
G(n_3) &= (\texttt{cons}, n_7, n_4),\\
G(n_4) &= (\texttt{nil}),\\
G(n_5) &= (\texttt{nil}),\\
G(n_6) &= (\texttt{nil}),\\
G(n_7) &= (\texttt{nil}).
\end{aligned}$$

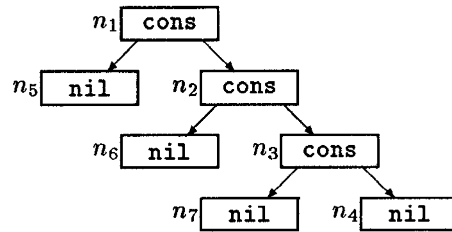

Figure 5: The storage graph for a list of three nils.

And if the value of variable x is the head of the list, then we have the storage state (G, f), with $f(x) = n_1$.

Here, we have chosen to make the four nils distinct nodes. If they are to be a single node (as they are in Lisp), then we will have

$$\begin{aligned}
G(n_1) &= (\texttt{cons}, n_4, n_2),\\
G(n_2) &= (\texttt{cons}, n_4, n_3),\\
G(n_3) &= (\texttt{cons}, n_4, n_4),\\
G(n_4) &= (\texttt{nil}).
\end{aligned}$$

In general, storage states may contain nodes that are inaccessible. Indeed, we can define the equivalent of the garbage collection function:

Definition 3 *$Obs((G, f))$ is the storage state (G', f), such that G' is the maximum subgraph of G reachable from $f[X]$. In other words, for every node in G', there exists a directed path from some $f(x)$ to that node, and G' is the largest of all such subgraphs.*

Of course, (G', f) always exists and is unique.

A program is a control-flow graph (S, E), where S is a set of vertices and E is a set of edges. Each vertex is a single statement. The input and output arities of a vertex depend on its statement type. Most statements have exactly one in-edge (called $in(s)$) and one out-edge (called $out(s)$). The exceptions are listed below. There is a distinguished start edge, e_0. As in the examples, each assignments of a variable is given a unique definition index (the x^d combination). For a given program, we call the set of all definitions D.

Each statement s computes a new storage state (G', f') from the input state (G, f):

$\texttt{pred}^t(x)$

A predicate has two out-edges, called $out_{true}(s)$ and $out_{false}(s)$.

Execution follows $out_{true}(s)$ if $type^G(f(x)) = t$, follows $out_{false}(s)$ otherwise. The storage state is unchanged:

$$(G', f') = (G, f).$$

$x^d \leftarrow y$

An assignment to x changes $f(x)$:

$$(G', f') = Obs((G, f\binom{x}{f(y)})).$$

$x^d \leftarrow \texttt{slot}_i^t(y)$

The behavior is undefined if $type^G(f(y)) \neq t$. Otherwise,

$$(G', f') = Obs((G, f\binom{x}{dest_i^G(f(y))})).$$

$x^d \leftarrow \textbf{cons}^t(y_1, \ldots, y_k)$

Let c be a new vertex (i.e., $c \notin |G|$). Indeed, to make the behavior deterministic, let c be the smallest vertex in $V - |G|$. Then,

$$\begin{aligned} G'' &= G\binom{c}{(t, f(y_1), \ldots, f(y_k))}, \\ f'' &= f\binom{x}{c}, \\ (G', f') &= Obs((G'', f'')). \end{aligned}$$

`join`

A `join` statement is where paths of control flow converge. It has an arbitrary number of inputs, named $in_1(s), \ldots, in_n(s)$, for an in-degree of n.

The actual operation is a no-op:

$$(G', f') = (G, f).$$

For the sake of determinism, the garbage collector (the function Obs) is explicitly invoked at every statement.

Since $\texttt{slot}_i^t(x)$ is undefined when $type(f(x)) \neq t$, we must always precede it with a type check:

Proposition 1 *Let vertex s_1 be the statement $x^d \leftarrow \texttt{slot}_i^t(y)$. The behavior of s_1 is always defined, if $in(s_1) = out_{true}(s_2)$ and s_2 is $\texttt{pred}^t(y)$.*

6 The Algorithm

For a program with variables X and definitions D, a *p-set* is a set of productions of the form $(u_0 \rightarrow tu_1 \ldots u_k)$, where the u_i's are nonterminals, t is a terminal taken from the set T of types, and k is the arity of t. A nonterminal is a set of definitions and variables: an element of $\mathcal{P}(D \cup X)$.

In the following discussion, when only selected parts of a production are of interest, we will often write it in an abbreviated form: $(u \rightarrow __)$, $(u \rightarrow t__)$, $(_ \rightarrow tu_1 \ldots u_k)$, and so on. For example, $p = (_ \rightarrow t__)$ means that p is of the form $(u_0 \rightarrow tu_1 \ldots u_k)$, where $u_0, \ldots, u_k$ are unrestricted.

Definition 4 *Let Q be a set of productions, and x a variable. The subset of Q* relative *to x is written $Q^{(x)}$ and is defined by*

$$Q^{(x)} = \{(u \rightarrow __) \in Q : x \in u\}.$$

We can now make precise the connection between storage states and p-sets. Let (G, f) be a storage state, Q a p-set, and d a function on $|G|$ to $\mathcal{P}(Q)$.

Definition 5 *We say (G, f) is a d-*instance *of Q (written $Q \overset{d}{\Rightarrow} (G, f)$), if for every $n \in |G|$, every $(u_0 \rightarrow tu_1 \ldots u_k) \in d(n)$, and every $x \in X$,*

(i) $t = type^G(n)$;

(ii) for all $1 \leq i \leq k$, there exists $q \in d(dest_i^G(n))$ such that $q = (u_i \rightarrow __)$;

(iii) if $n = f(x)$ then $d(n)^{(x)}$ is a singleton set;

(iv) if $n \neq f(x)$ then $d(n)^{(x)} = \emptyset$.

We denote the set of all instances of Q by $Inst(Q)$.[4]

To account for shared structures, we allow more than one production to derive the same node in an instance (i.e., $d(n)$ is a set).

For example, given storage state (G, f), with G as shown in Figure 5 and $f(x) = n_1$, and given p-set

$$\begin{aligned} Q = \{&\{x, x^1\} \rightarrow \textbf{cons}\ \emptyset\ \emptyset \mid \textbf{nil} \mid \textbf{other}, \\ &\emptyset \rightarrow \textbf{cons}\ \emptyset\ \emptyset \mid \textbf{nil} \mid \textbf{other}\}, \end{aligned}$$

then $Q \overset{d}{\Rightarrow} (G, f)$ with

$$\begin{aligned} d(n_1) &= \{\{x, x^1\} \rightarrow \textbf{cons}\ \emptyset\ \emptyset\}, \\ d(n_2) &= \{\emptyset \rightarrow \textbf{cons}\ \emptyset\ \emptyset\}, \\ d(n_3) &= \{\emptyset \rightarrow \textbf{cons}\ \emptyset\ \emptyset\}, \\ d(n_4) &= \{\emptyset \rightarrow \textbf{nil}\}, \\ d(n_5) &= \{\emptyset \rightarrow \textbf{nil}\}, \\ d(n_6) &= \{\emptyset \rightarrow \textbf{nil}\}, \\ d(n_7) &= \{\emptyset \rightarrow \textbf{nil}\}. \end{aligned}$$

Definition 6 *If r is a function mapping nonterminals to nonterminals, then $\hat{r}$ is the extension of r to productions:*

$$\hat{r}((u_0 \rightarrow tu_1 \ldots u_k)) = (r(u_0) \rightarrow tr(u_1) \ldots r(u_k)).$$

The domain of $\hat{r}$ is the set of all productions that can be formed from the domain of r and the set of types T.

[4] We use the term *instance* instead of *derivation* because our notion is not quite the standard one.

For flow graph (S, E), variables X, and definitions D, the analysis produces a function P on E. For each program point $e \in E$, $P(e)$ is a p-set.

The algorithm finds a solution for P given the statement-specific flow equations (stated below) and the initial condition at the entry edge, e_0:

$$P(e_0) = \left[\bigcup_{x \in X} U(\{x, x^0\})\right] \cup U(\emptyset),$$

where

$$U(u) = \{(u \rightarrow t\overbrace{\emptyset \ldots \emptyset}^{k}) : t \in T, t \text{ is } k\text{-ary}\}.$$

For example, if we have variables x_1, x_2, x_3 and types t_1, t_2 (of arities 0 and 1, respectively), then

$$\begin{aligned} P(e_0) = \{&\{x_1, x_1^0\} \rightarrow t_1 \mid t_2\emptyset, \\ &\{x_2, x_2^0\} \rightarrow t_1 \mid t_2\emptyset, \\ &\{x_3, x_3^0\} \rightarrow t_1 \mid t_2\emptyset, \\ &\emptyset \rightarrow t_1 \mid t_2\emptyset\}. \end{aligned}$$

The algorithm we actually use is a forward propagation. At each flow-graph node, the per-statement flow equation is used to compute the p-sets at the exits from the p-sets at the entries. Initially, $P(e) = \emptyset$ for all $e \in E$ except e_0.

In the following, where the vertex s of interest is understood, we will use the shorthand P_{in} to mean $P(in(s))$, P_{out} to mean $P(out(s))$. These are the per-statement equations:

$\mathtt{pred}^t(x)$

A predicate is a filter. First, we define the set Q of productions in ${P_{in}}^{(x)}$ that satisfy the predicate, and its complement Q':

$$\begin{aligned} Q &= \{p \in {P_{in}}^{(x)} : p = (_ \rightarrow t__)\}, \\ Q' &= {P_{in}}^{(x)} - Q. \end{aligned}$$

Then,

$$\begin{aligned} P_{true} &= \begin{cases} \emptyset & \text{if } Q = \emptyset \\ P_{in} - Q' & \text{otherwise,} \end{cases} \\ P_{false} &= \begin{cases} \emptyset & \text{if } Q' = \emptyset \\ P_{in} - Q & \text{otherwise.} \end{cases} \end{aligned}$$

Here, we can see how the *rounding-to-$\emptyset$* rule works. For P_{true}, if Q is empty, then $Q' = {P_{in}}^{(x)}$ and $(P_{in} - Q')^{(x)} = \emptyset$. Therefore, we have

$$Inst(P_{in} - Q') = \emptyset = Inst(\emptyset).$$

This is not the only case in which productions can be deleted from a p-set. Section 8 discusses two general procedures for p-set simplification.

$x^d \leftarrow y$

We first define a function r on nonterminals, then P_{out} itself:

$$\begin{aligned} r(u) &= \begin{cases} u \cup \{x, x^d\} & \text{if } y \in u \\ u - \{x\} & \text{otherwise,} \end{cases} \\ P_{out} &= \hat{r}[P_{in}]. \end{aligned}$$

$x^d \leftarrow \mathtt{slot}_i^t(y)$

This step is only defined when all productions in ${P_{in}}^{(y)}$ are of the form $(_ \rightarrow t__)$.

$$\begin{aligned} Q &= {P_{in}}^{(y)}, \\ R &= P_{in} - {P_{in}}^{(y)}, \\ P_{out} &= \bigcup_{q \in Q} A_q. \end{aligned}$$

Each A_q is a set defined in terms of q and R. Let $q = (u_0 \rightarrow tu_1 \ldots u_k)$. If $u_i \cap X \neq \emptyset$ then

$$\begin{aligned} A_q &= \hat{a}[R \cup \{q\}], \\ a(v) &= \begin{cases} v \cup \{x, x^d\} & \text{if } v = u_i \\ v - \{x\} & \text{otherwise.} \end{cases} \end{aligned}$$

If $u_i \cap X = \emptyset$ then

$$\begin{aligned} A_q &= \hat{b}[R] \cup r[\hat{b}[\{q\}]] \cup l[\hat{b}[S]], \\ S &= \{p \in P_{in} : p = (u_i \rightarrow __)\}, \end{aligned}$$

and

$$\begin{aligned} b(v) &= v - \{x\}, \\ r((v_0 \rightarrow tv_1 \ldots v_k)) &= \\ &(v_0 \rightarrow tv_1 \ldots v_i \cup \{x, x^d\} \ldots v_k), \\ l((v_0 \rightarrow t'v_1 \ldots v_\ell)) &= \\ &(v_0 \cup \{x, x^d\} \rightarrow t'v_1 \ldots v_\ell). \end{aligned}$$

$x^d \leftarrow \mathtt{cons}^t(y_1, \ldots, y_k)$

$$\begin{aligned} r(u) &= u - \{x\}, \\ Q &= \{(\{x, x^d\} \rightarrow tr(u_1) \ldots r(u_k)) : \\ &\quad \text{for every } 1 \leq i \leq k \\ &\quad \text{some } (u_i \rightarrow __) \in {P_{in}}^{(y_i)}\}, \\ P_{out} &= \hat{r}[P_{in}] \cup Q. \end{aligned}$$

$\mathtt{join}$

A join node is just the union of all inputs.

$$P_{out} = \bigcup_i P_{in_i}.$$

We again need a theorem on the effectiveness of type checking:

Proposition 2 *Let vertex s_1 be the statement $x^d \leftarrow \mathtt{slot}_i^t(y)$. Then the flow equation for s_1 is always defined, if $in(s_1) = out_{true}(s_2)$ and s_2 is $\mathtt{pred}^t(y)$.*

7 Correctness

Given a program, we can also define $M(e)$ to be the set of all possible run-time storage states at $e \in E$. Specifically, a state (G, f) is in $M(e)$ if and only if there exists some execution of the program (starting at the entry with some initial storage state) that produces the state (G, f) when control reaches e.

The algorithm is correct if for all $e \in E$ and $x \in X$,

$$T_{runtime}(e, x) \subset T_{predicted}(e, x),$$

where

$$\begin{aligned} T_{runtime}(e, x) &= \{type^G(f(x)) : (G, f) \in M(e)\}, \\ T_{predicted}(e, x) &= \{t : (_ \to t__) \in P(e)^{(x)}\}. \end{aligned}$$

We can, however, state a stronger theorem:

Theorem 1 (Correctness) *At every program point e, $M(e) \subset Inst(P(e))$.*

The algorithm's precision is measured by $Inst(P(e)) - M(e)$.

With the machinery developed so far, the proof of Theorem 1 is straightforward, but too long to include in this paper.

8 Simplification and Approximation

As stated, the basic algorithm will sometimes introduce unused productions into the generated p-sets. These stray productions can, at best, slow down the analysis and, at worst, lead to imprecision. For example, consider this sequence of statements:

```
x¹ ← nil
y¹ ← nil
if C goto L1
y² ← cons(y, y)
x² ← cons(y, y)
L1:
    {{x,x¹} → nil
     {x,x²} → cons {y,y²} {y,y²}
     {y,y¹} → nil
     {y,y²} → cons {y¹} {y¹}
     {y¹} → nil}
if consp(y) goto error
L2:
    {{x,x¹} → nil
     {x,x²} → cons {y,y²} {y,y²}
     {y,y¹} → nil
     {y¹} → nil}
if consp(x) ...
    {{x,x²} → cons {y,y²} {y,y²}
     {y,y¹} → nil
     {y¹} → nil}
    {{x,x¹} → nil
     {y,y¹} → nil
     {y¹} → nil}
```

If the condition C is unknown at compile time, we get the p-sets as shown. At label **L2**, since **y** cannot be a cons, **x** cannot be one either. The final conditional should always fail. The analysis, however, fails to predict this.

There is a simple and efficient procedure for eliminating such obviously unusable productions. We state it as a theorem:

Theorem 2 (Bottom-up Simplification) *Given a p-set Q, $p \in Q$, and $p = (u_0 \to tu_1 \ldots u_k)$, if for some u_i there exists no production of the form $(u_i \to ___)$ in Q, then $Inst(Q - \{p\}) = Inst(Q)$.*

Another class of unused productions is illustrated by the last three occurrences of $(\{\mathtt{y}^1\} \to \mathtt{nil})$ in the above example. The production serves no purpose in those p-sets, because it is never used to generate any observable storage state:

Theorem 3 (Top-down Simplification) *Given Q, $p \in Q$, $p = (u \to ___)$, and $u \cap X = \emptyset$, if u does not occur on the right-hand side of any production in Q, then $Obs[Inst(Q - \{p\})] = Obs[Inst(Q)]$.*

Because nonterminals are drawn from the set $\mathcal{P}(X \cup D)$, there can be exponentially many of them. Indeed, the basic algorithm is exponential in the worst case, in both space and time. We can, however, keep the algorithm well-behaved by sacrificing some of the precision. Again, we state this as a theorem:

Theorem 4 (Approximation) *Let Q be a p-set, and r a function on nonterminals to nonterminals. If $r(u) \cap X = u \cap X$ for all u, then $Inst(Q) \subset Inst(\hat{r}[Q])$.*

In other words, we can rename nonterminals as long as the variables in them are preserved. Some of the precision will be lost, but the result remains correct.

Theorem 4 can be applied to the intermediate p-sets in the running algorithm, if r is chosen judiciously to preserve monotonicity. Specifically, because r need not be one-to-one, the theorem can be used to reduce the number of distinct nonterminals within a p-set, thus reducing its size. This procedure can only affect the definitions; it cannot reduce the number of variables in a nonterminal. There is indeed an orthogonal method that can deal with the variables. However, we have found Theorem 4 to be sufficient in practice. For example, simply limiting the number of definitions in a nonterminal to one leads to good performance with acceptable precision.

Thus we can say that Theorem 4 generates a family of algorithms, each at a different complexity-precision tradeoff.

9 Final Comments

A first implementation of the algorithm exists. It accepts a subset of Common Lisp, and produces an annotated program.

Selective update ($\texttt{slot}_i^t(x) \leftarrow y$) can be handled by the algorithm. However, in the most naive implementation, it may affect every production of the same type. An algorithm can do better if it can answer these questions for every pair of productions in a p-set:

- Can they map to the same node in some instance?
- Can they map to different nodes in some instance?

The first question is exactly the sharing information sought by alias analysis. Just as Baker has shown that Milner's algorithm can produce sharing information [1], we believe p-sets can be similarly augmented. Even without modification, productions introduced by two different `cons` statements can never map to the same node in an instance. We believe this information can be of comparable quality to that provided by the algorithm of Chase, Wegman, and Zadeck [3]. How this interacts with p-set approximation in a real implementation is yet to be seen.

References

[1] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 218-226, June 1990.

[2] Randall D. Beer. Preliminary report on a practical type inference system for Common Lisp. *Lisp Pointers*, 1(2):5–12, June-July 1987.

[3] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[4] Carsten K. Gomard. Partial type inference for untyped functional programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 282–287, June 1990.

[5] Philip Johnson. *Type Flow Analysis for Exploratory Software Development.* PhD thesis, University of Massachusetts, Amherst, September 1990.

[6] Neil D. Jones and Steven S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. In *Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 77–94, January 1976.

[7] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[8] Marc A. Kaplan and Jeffrey D. Ullman. A general scheme for the automatic inference of variable types. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 60–75, January 1978.

[9] Marc A. Kaplan and Jeffrey D. Ullman. A scheme for the automatic inference of variable types. *Journal of the Association for Computing Machinery*, 27(1):128–145, January 1980.

[10] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[11] Kwan-Liu Ma and Robert R. Kessler. TICL—a type inference system for Common Lisp. *Software—Practice and Experience*, 20(6):593–623, June 1990.

[12] Robert A. MacLachlan, April 1991. Private communication.

[13] Robert A. MacLachlan, editor. CMU Common Lisp user's manual. Technical Report CMU-CS-91-108, Carnegie-Mellon University, February 1991.

[14] Terrence C. Miller. Type checking in an imperfect world. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 237–243, January 1979.

[15] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[16] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21, January 1985.

[17] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.