Inferring Recursive Data Types

David McAllester

AT&T Labs, 600 Mountain Ave, Murray Hill N.J. 07974, dmac@research.att.com, (908)582-5412, http::/www.ai.mit.edu/people/dam/dam.html

Abstract. This paper contains five results on the problem of inferring types. The first is that type inference over recursive types with unions and data constructors can be done in cubic time using a flow analysis. The second is a general theorem characterizing the time complexity of bottom-up logic programs. The $O(n^3)$ running time of the flow analysis is a corollary of this bottom-up run time theorem. The third is that for shallow case statements typability by the semantic types of Aiken, Wimmers and Lakshman is equivalent to typability by recursive types and hence can be determined by flow analysis. The fourth is that, even for first order programs of arity one, typability by recursive types is PSPACE hard for polymorphic programs. The final result is that for any fixed bound on order and arity Hindley-Milner typability can be determined in pseudo-linear time, i.e., $O(n\alpha(n))$ where α is the inverse Ackerman function. The last two results suggest that let-polymorphism over simple types is fundamentally more tractable than let-polymorphism over recursive types.

1 Introduction

There are a variety of applications where it is useful to be able to determine that certain syntactic expressions have certain semantic properties. Type systems provide effective procedures for establishing semantic properties, e.g., that a certain expression always denotes an integer. A type *checking* system simply verifies that a given expression has a given property. A type *inference* system takes a given expression and "synthesizes" or "invents" semantic properties that hold of it. Type inference has a variety of applications in compilers and in automated reasoning systems. Here we consider the problem of of inferring (or inventing) properties which can be expressed by recursive types involving unions and data constructors.

The problem of inferring recursive types can be understood in the context of the more traditional Hindley-Milner type inference system [21, 11]. In the Hindley-Milner system data constructors must have user-specified types such as the following.

$$\mathtt{NIL}: \forall\sigma\ \ (\sigma)\mathtt{list}, \qquad \mathtt{CONS}: \forall\sigma\ \ \sigma \times (\sigma)\mathtt{list} \rightarrow (\sigma)\mathtt{list}$$

Here we are interested in the problem of declaration-free type inference, i.e., the problem of constructing types, such as the notion of a list of integers, from a program without declarations for data constructors such as CONS. Declaration-free type inference, and the synthesis of recursive types, has been explored both in the context of logic programming, e.g., [22, 14, 10, 16], and in the context of functional programming [23, 1, 2, 12]. Declaration-free type inference has two advantages over Hindley-Milner types. First, it can be used for languages, such as Scheme, where type declarations are not provided. Second it allows the same data constructor to be used in a variety of incompatible ways in different parts of the same program. For example, it allows one to use CONS as both a list and tree constructor. As discussed below, declaration-free type inference has the disadvantage of taking longer to perform.

This paper contains five results associated with the inference of recursive types over unions and data constructors.

1. For monomorphic programs with shallow case statements, type inference for recursive types with unions and data constructors can be done in cubic time using a flow analysis.
2. The cubic time performance of the analysis can be derived from a more general theorem — the deductive closure of any set of bottom-up bound inference rules can be computed in time proportional to the size of the initial data base plus the number of "prefix firings" of the rule set.
3. For programs with shallow case statements, the semantic type system of Aiken, Wimmers and Lakshman [2] types the same programs as those typable by the system of recursive types mentioned above. The theorem shows that Aiken-Wimmers-Lakshman approach gives essentially the same results (on monomorphic programs) as the Heintze approach [12].
4. Even for first order programs of arity one and shallow case statements, typability by recursive types with unions and data constructors is PSPACE hard for let-polymorphic programs.
5. For any fixed bound on order and arity traditional Hindley-Milner typability can be determined in pseudo-linear time, i.e., $O(n\alpha(n))$ where α is the inverse Ackerman function. This suggests that traditional Hindley-Milner polymorphic type inference is fundamentally more tractable than polymorphic type inference for recursive types.

The type inference procedure based on flow analysis is related to the set based analysis methods of Heintze [12]. The analysis is also similar to that of Shivers [27] or Jagannathan and Write [17] but includes data constructors and union types. It can also be viewed as an extension of the recently established equivalence between flow analysis and recursive types in the absence of data constructors [5, 25, 13, 26]. Here we formulate flow analysis as a process of computing a directed graph on the subexpressions of the input where $u \to w$ means that w is a "possible value" of u. A program is typable by flow analysis if the computed flow graph satisfies certain natural safety conditions. The flow analysis is a type inference process in the sense that a recursive type for each program expression be easily constructed given the final flow graph.

The flow analysis is defined by inference rules which are intended to be used in a forward chaining (bottom-up) manner — the rules derive directed arcs on the subexpressions of the input program and the final flow graph is the closure of the set of arcs under these rules. There has been considerable interest in the database community in finding methods of efficiently computing deductive closures, i.e., in bottom-up logic programming [24]. The second result of this paper provides a general way of "reading off" the order of run time of an algorithm defined by a set of forward chaining inference rules, i.e., a bottom-up logic program. The cubic time performance of the flow analysis underlying the first result is derived as an application of the more general theorem.

The third result relates the flow analysis type inference algorithm to the semantic type system given by Aiken, Wimmers and Lakshman [1, 2]. They take an ideal model of types [19] in which semantic types are defined to be certain subsets of a semantic domain. A program is typable by semantic types if every subexpression can be assigned a semantic type satisfying certain constraints. Most notably, for each application $(f\ e)$ we must have that f is assigned a function type $\alpha \rightarrow \beta$ where α contains (but need not equal) the type of e. The semantic view of type inference has, to a large extent, motivated recent work on solving systems of semantic set constraints [3, 7, 4, 8, 9]. Here we show that for monomorphic programs with shallow case statements typability in the Aiken-Wimmers-Lakshman system is equivalent to typability by the flow analysis discussed above. For monomorphic programs with deep case statements typability in the Aiken-Wimmers-Lakshman system can be shown to be EXPTIME complete, although that result is beyond the scope of this paper.

The fourth result concerns the difficulty of let-polymorphism. Here we consider a program containing let expressions to be polymorphically typable if the let-expansion of that program is monomorphically typable. We show that even for first order programs of arity one with shallow case statements the problem of determining let-polymorphic typability is PSPACE hard.

The fifth result demonstrates a contrast between the difficulty of let-polymorphism for recursive types and let-polymorphism for simple types. Henglein has shown that, for any given bound b on the size of types, the problem of determining Hindley-Milner typability by types of size b is polynomial time decidable [15]. This immediately implies that for any bound on order and arity Hindley-Milner typability is polynomial time decidable. Here we give a more refined result — typability by types of size b can be determined in $O(bn\alpha(bn))$ time where α is the inverse Ackerman function. More concisely, for bounded order and arity, Hindley-Milner typability is decidable in pseudo-linear time.

Although polymorphic typability for bounded order and arity is PSPACE hard in Aiken-Wimmers-Lakshman system, it may still be possible to achieve good performance in practice by generalizing the Hindley-Milner type inference procedure. This can be done by using "constrained types" rather than the traditional type schemes. Constrained types are used by Aiken, Wimmers and Lakshman and have also been studied in the context of recursive types by Palsberg and Smith [26]. Constrained types are beyond the scope of this paper.

2 Type Inference by Flow Analysis

We consider terms (programs) defined by the following grammar.

$$\begin{aligned} e ::= {} & x \mid (e_1\ e_2) \mid \lambda x.e \mid c(e_1,\ \ldots,\ e_n) \\ & \mid \mathtt{CASE}(e_1 \ \mathtt{of}\ c(x_1,\ \ldots,\ x_n)\mathtt{:}e_2,\ y\mathtt{:}e_3) \\ & \mid \mathrm{LET}\ x = e_1\ \mathtt{in}\ e_2 \end{aligned}$$

In case expressions the pattern variables $x_1, \ldots, x_n$ must all be distinct. A term is called *monomorphic* if it does not contain any let-expressions. In the presence of recursive types we can type self application, i.e., $(x\ x)$. This allows us to use a call by value variant of the traditional λ-calculus Y operator to model recursion.

We give a standard call by value semantics based on rewrite rules for monomorphic programs (let-expressions are discussed in section 8). First we define a *rewrite value* to be either a λ-expression or an application of a constructor to values. The phrase "rewrite value" is used to distinguish these values from the "environment values" described below. The rules given below are to be applied under a rightmost outermost strategy subject to the restriction that v ranges over rewrite values and that reduction not be done inside λ-expressions. The value of a term is its normal form under this reduction strategy if such a normal form exists and $\perp$ otherwise.

$$((\lambda x.e)\ \ v) \to e[v/x]$$

$$\mathtt{CASE}(c(v_1,\ \ldots,\ v_n)\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n)\mathtt{:}e_2,\ y\mathtt{:}e_3) \to e_2[v_1/x_1,\ \ldots,\ v_n/x_n]$$

$$\mathtt{CASE}(v\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n)\mathtt{:}e_2,\ y\mathtt{:}e_3) \to e_3[v/y]\ \ v \text{ not an application of } c$$

$$(c(v_1,\ \ldots,\ v_n)\ v) \to \mathtt{ERROR}$$

$$u \to \mathtt{ERROR}\ \ \text{if } u \text{ contains } \mathtt{ERROR}$$

A term e is called *safe* if it has a value other than ERROR (nonterminating terms are safe). Safety is undecidable in general. However there are various type systems having the property that the typable terms are safe and where typability is decidable. For any such type system there will be safe terms which are untypable. Type systems trade off efficiency and accuracy — type inference is generally more difficult in type systems which allow more accurate types.

Figure 1 gives an "environment evaluator". The evaluator is defined with bottom-up (forward chaining) inference rules. To understand the notations used in the rules some additional definitions are needed. For any term e we define $FV(e)$ to be the set of free variables of e. An *environment value* is either a term of the form $c(v_1,\ \ldots,\ v_n)$ where $v_1, \ldots, v_n$ are environment values or a "closure" of the form $\langle \lambda x.u,\ \rho\rangle$ where ρ is a variable environment mapping $FV(\lambda x.u)$ to environment values.

The base case is given by constants, such as NIL, and closed λ-expressions. In the inference rules v is restricted to range over environment values. For any variable environment σ mapping variables to environment values, and any set V of variables, we define $\sigma|_V$ to be the restriction of the environment σ to the variables in V.

LAM
$$\frac{\texttt{EVAL}(\lambda x.u,\ \rho)}{\langle \lambda x.u,\ \rho\rangle \rightarrow \langle \lambda x.u,\ \rho|_{FV(\lambda x.u)}\rangle}$$

BETA1
$$\frac{\texttt{EVAL}((f\ w),\ \rho)}{\texttt{EVAL}(f,\ \rho),\quad \texttt{EVAL}(w,\ \rho)}$$

BETA2
$$\frac{\begin{array}{l}\texttt{EVAL}((f\ w),\ \rho)\\ \langle f,\ \rho\rangle \rightarrow \langle \lambda x.u,\ \sigma\rangle\\ \langle w,\ \rho\rangle \rightarrow v\end{array}}{\langle (f\ w),\ \rho\rangle \rightarrow \langle u,\ \sigma[x := v]\rangle}$$

BETA3
$$\frac{\begin{array}{l}\texttt{EVAL}((f\ w),\ \rho)\\ \langle f,\ \rho\rangle \rightarrow c(v_1,\ \ldots,\ v_n)\end{array}}{\texttt{ERROR}}$$

VAR
$$\frac{\texttt{EVAL}(x,\ \rho)}{\langle x,\ \rho\rangle \rightarrow \rho(x)}$$

SPROP
$$\frac{\begin{array}{l}\texttt{EVAL}(u,\ \rho)\\ \langle u,\ \rho\rangle \rightarrow \langle w,\ \sigma\rangle\end{array}}{\texttt{EVAL}(w,\ \sigma)}$$

TRANS
$$\frac{\begin{array}{l}\langle u,\ \rho\rangle \rightarrow \langle w,\ \sigma\rangle\\ \langle w,\ \sigma\rangle \rightarrow v\end{array}}{\langle u,\ \rho\rangle \rightarrow v}$$

CONS1
$$\frac{\texttt{EVAL}(c(e_1,\ \ldots,\ e_n),\ \rho)}{\texttt{EVAL}(e_1,\ \rho),\ \ldots,\ \texttt{EVAL}(e_n,\ \rho)}$$

CONS2
$$\frac{\begin{array}{l}\texttt{EVAL}(c(e_1,\ \ldots,\ e_n),\ \rho)\\ \langle e_1,\ \rho\rangle \rightarrow v_1\\ \vdots\\ \langle e_n,\ \rho\rangle \rightarrow v_n\end{array}}{\langle c(e_1,\ \ldots,\ e_n),\ \rho\rangle \rightarrow c(v_1,\ \ldots,\ v_n)}$$

CASE1
$$\frac{\texttt{EVAL}(\texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho)}{\texttt{EVAL}(e_1,\ \rho)}$$

CASE2
$$\frac{\begin{array}{l}\texttt{EVAL}(\texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho)\\ \langle e_1,\ \rho\rangle \rightarrow c(v_1,\ \ldots,\ v_n)\end{array}}{\begin{array}{l}\langle \texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho\rangle\\ \rightarrow \langle e_2,\ \rho[x_1 := v_1, \ldots,\ x_n := v_n]\rangle\end{array}}$$

CASE3
$$\frac{\begin{array}{l}\texttt{EVAL}(\texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho)\\ \langle e_1,\ \rho\rangle \rightarrow c'(v_1,\ \ldots,\ v_m)\\ c' \neq c\end{array}}{\begin{array}{l}\langle \texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho\rangle\\ \rightarrow \langle e_2,\ \rho[y := c'(v_1,\ \ldots,\ v_m)]\rangle\end{array}}$$

CASE4
$$\frac{\begin{array}{l}\texttt{EVAL}(\texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho)\\ \langle e_1,\ \rho\rangle \rightarrow \langle \lambda x.u,\ \sigma\rangle\end{array}}{\begin{array}{l}\langle \texttt{CASE}(e_1\ \texttt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3),\ \rho\rangle\\ \rightarrow \langle e_2,\ \rho[y := \langle \lambda x.u,\ \sigma\rangle]\rangle\end{array}}$$

Fig. 1. An Operational Semantics. An environment value is either a "closure" $\langle \lambda x.u\ \rho|_{FV(\lambda x.u)}\rangle$ where ρ maps variables to values or a data structure $c(v_1,\ \ldots,\ v_n)$ where each v_i is a value. Note that closed λ-expressions and constants such as `NIL` provide base cases. In all rules v and v_i range over environment values and all environments map variables to values. The rules are intended to be run in a forward chaining or "bottom-up" fashion.

$$\text{EVAL}\qquad \frac{\mathtt{EVAL}(e)}{\mathtt{EVAL}(u)\ \ u \text{ a subterm of } e}$$

$$\text{IDENT}\qquad \frac{\mathtt{EVAL}(u)}{u \to u}$$

$$\text{LAM}\qquad \frac{\mathtt{EVAL}(\lambda x.u)}{\exists\, \lambda x.u}$$

$$\text{BETA2}\qquad \frac{\begin{array}{l}\mathtt{EVAL}((e_1\ e_2))\\ e_1 \to \lambda x.u\\ \exists\, e_2\end{array}}{x \to e_2,\ \ (e_1\ e_2) \to u}$$

$$\text{BETA3}\qquad \frac{\begin{array}{l}\mathtt{EVAL}((e_1,\ e_2)),\ \ e_1 \to c(u_1,\ \ldots,\ u_n)\\ \exists\, c(u_1,\ \ldots,\ u_n)\end{array}}{\mathtt{ERROR}}$$

$$\text{CONS}\qquad \frac{\begin{array}{l}\mathtt{EVAL}(c(u_1,\ \ldots,\ u_n))\\ \exists\, u_1,\ \ldots,\ \exists\, u_n\end{array}}{\exists\, c(u_1,\ \ldots,\ u_n)}$$

$$\text{TRANS1}\qquad \frac{e \to u,\ \ \exists\, u}{\exists\, e}$$

$$\text{TRANS2}\qquad \frac{e \to u,\ \ u \to w}{e \to w}$$

$$\text{CASE2}\qquad \frac{\begin{array}{l}\mathtt{EVAL}(\mathtt{CASE}(e_1\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3))\\ e_1 \to c(u_1,\ \ldots,\ u_n),\ \ \exists\, c(u_1,\ \ldots,\ u_n)\end{array}}{\begin{array}{l}x_1 \to u_1,\ \ldots,\ x_n \to u_n\\ \mathtt{CASE}(e_1\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3) \to e_2\end{array}}$$

$$\text{CASE3}\qquad \frac{\begin{array}{l}\mathtt{EVAL}(\mathtt{CASE}(e_1\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3))\\ e_1 \to \lambda x.u\end{array}}{\begin{array}{l}y \to \lambda x.u\\ \mathtt{CASE}(e_1\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3) \to e_3\end{array}}$$

$$\text{CASE4}\qquad \frac{\begin{array}{l}\mathtt{EVAL}(\mathtt{CASE}(e_1\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3))\\ e_1 \to c'(u_1,\ \ldots,\ u_n)\ \ \ c' \neq c\\ \exists\, c'(u_1,\ \ldots,\ u_n)\end{array}}{\begin{array}{l}y \to c'(u_1,\ \ldots,\ u_n)\\ \mathtt{CASE}(e_1\ \mathtt{of}\ c(x_1,\ \ldots,\ x_n){:}e_2,\ y{:}e_3) \to e_3\end{array}}$$

Fig. 2. Rules for Flow Analysis.

We let $\vdash_E$ be the inference relation defined by the environment evaluation rules so that $\Sigma \vdash_E \Phi$ means that the assertion Φ is provable from the set of assertions Σ using the rules given in figure 1. Every environment value v can be uniquely converted to a rewrite value $R(v)$ by replacing each closure of the form $\langle \lambda x.u,\ \rho\rangle$ by $\rho'(\lambda x.u)$ where ρ' is the substitution such that $\rho'(y) = R(\rho(y))$. The environment rules implement the rewrite rule semantics in the sense that $\mathtt{EVAL}(e,\ \emptyset) \vdash_E \langle e,\ \emptyset\rangle \to v$ if and only if e rewrites to $R(v)$ under the rewrite rules given above with the given rewriting strategy. The environment evaluator satisfies the invariants that if $\mathtt{EVAL}(e,\ \emptyset) \vdash_E \mathtt{EVAL}(u,\ \rho)$ then u is a subterm of e and, furthermore, any λ-expression contained in a closure appearing ρ is also a subterm of e.

Figure 2 gives flow analysis inference rules. The basic idea is to "abstract" the environment evaluation assertions of the form $\langle u,\ \rho\rangle \to \langle w,\ \sigma\rangle$ to "flow assertions" of the form $u \to w$. It is useful to note the structural similarities between the rules in figures 1 and 2 — analogous rules have been given the same name where possible. We let $\vdash_F$ be the inference relation defined by the rules in figure 2. Intuitively, the flow rules "over-generate" the rewrite relation so that if $\mathtt{EVAL}(e,\ \emptyset)\ \vdash_E$ ERROR then $\mathtt{EVAL}(e)\ \vdash_F$ ERROR. We argue more carefully below that if $\mathtt{EVAL}(e)\ \not\vdash_F$ ERROR then e is safe. Since the flow rules only generate rewrite relations between subterms of the input, flow analysis terminates in polynomial time. So we have a polynomial time analysis which can guarantee safety.

As an example, let Y stand for $\lambda f.((\lambda h.\ (h\ h))\ (\lambda h.\ (f\ (\lambda v.\ ((h\ h)\ v)))))$ and let APPEND be $(Y\ \lambda\mathrm{app}.\ \lambda xy.\ \mathtt{CASE}(x\ \mathtt{of}\ \mathtt{CONS}(z,\ w)\mathtt{:CONS}(z,\ (\mathrm{app}\ w\ y)),\ u\mathtt{:}y))$. Here we have used $\lambda xy.e$ as an abbreviation for $\lambda x.\lambda y.e$ and $(f\ u\ v)$ as an abbreviation for $((f\ u)\ v)$. Now consider the input $(\mathtt{APPEND}\ \mathtt{CONS}(a,\ \mathtt{CONS}(b,\ \mathtt{NIL})),\ \mathtt{NIL})$ where a and b are constructor constants. The flow rules generate a variety of flow arcs including the following.

$$(\mathtt{APPEND}\ \mathtt{CONS}(a,\ \mathtt{CONS}(b,\ \mathtt{NIL})),\ \mathtt{NIL}) \to \mathtt{CASE}(x\ \mathtt{of}\ \mathtt{CONS}(z,\ w)\mathtt{:CONS}(z,\ (\mathrm{app}\ w\ y)),\ u\mathtt{:}y)$$

$$x \to \mathtt{CONS}(a,\ \mathtt{CONS}(b,\ \mathtt{NIL})) \quad y \to \mathtt{NIL} \quad z \to a \quad w \to \mathtt{CONS}(b,\ \mathtt{NIL})$$

$$\mathtt{CASE}(x\ \mathtt{of}\ \mathtt{CONS}(z,\ w)\mathtt{:CONS}(z,\ (\mathrm{app}\ w\ y)),\ u\mathtt{:}y) \to \mathtt{CONS}(z,\ (\mathrm{app}\ w\ y))$$

$$\mathrm{app} \to \lambda v.((h\ h)\ v)$$

$$(h\ h) \to \lambda xy.\ \mathtt{CASE}(x\ \mathtt{of}\ \mathtt{CONS}(z,\ w)\mathtt{:CONS}(z,\ (\mathrm{app}\ w\ y)),\ u\mathtt{:}y)$$

$$(\mathrm{app}\ w\ y) \to \mathtt{CASE}(x\ \mathtt{of}\ \mathtt{CONS}(z,\ w)\mathtt{:CONS}(z,\ (\mathrm{app}\ w\ y)),\ u\mathtt{:}y)$$

$$x \to \mathtt{CONS}(b,\ \mathtt{NIL}) \quad z \to b \quad w \to \mathtt{NIL}$$

$$(\mathrm{app}\ w\ y) \to \mathtt{CONS}(z,\ (\mathrm{app}\ w\ y))$$

$$(\mathrm{app}\ w\ y) \to \mathtt{NIL}$$

Note that each flow arc is between subterms of the input. Flow arcs of the form $e \to c(u_1,\ \ldots,\ u_n)$ can be viewed as productions in a regular grammar. For example consider the following subset of the above arcs.

$$(\mathrm{app}\ w\ y) \to \mathtt{NIL} \quad (\mathrm{app}\ w\ y) \to \mathtt{CONS}(z,\ (\mathrm{app}\ w\ y))$$

$$z \to a \quad z \to b$$

These arcs can be viewed as defining the following grammar where X represents the type of $(\mathrm{app}\ w\ y)$ and Z represents the type of the variable z.

$$X ::= \mathtt{NIL} \mid \mathtt{CONS}(Z,\ X)$$

$$Z ::= a \mid b$$

Now we prove more carefully that if $\mathtt{EVAL}(e)\ \not\vdash_F$ ERROR then e is safe. It suffices to show that if $\mathtt{EVAL}(e,\ \emptyset)\ \vdash_E$ ERROR then $\mathtt{EVAL}(e)\ \vdash_F$ ERROR. To do this we first create an extended inference relation $\vdash_{F'}$ defined by adding the following two "value rules" to the flow rules of figure 2.

VAL1

$$\frac{\begin{array}{l}\texttt{EVAL}(c(u_1,\ \ldots,\ u_n)) \\ u_1 \Rightarrow v_1 \\ \vdots \\ u_n \Rightarrow v_n\end{array}}{c(u_1,\ \ldots,\ u_n) \Rightarrow c(v_1,\ \ldots,\ v_n)}$$

VAL2

$$\frac{\begin{array}{l}\texttt{EVAL}(\lambda x.u) \\ FV(\lambda x.u) = \{y_1,\ \ldots,\ y_n\} \\ y_1 \Rightarrow v_1 \\ \vdots \\ y_n \Rightarrow v_n\end{array}}{\lambda x.u \Rightarrow \langle \lambda x.u,\ \{y_1 \mapsto v_1,\ \ldots,\ y_n \mapsto v_n\}\rangle}$$

VAL3

$$\frac{\begin{array}{l}u \rightarrow w \\ w \Rightarrow v\end{array}}{u \Rightarrow v}$$

The base case for these rules is provided by constructors of no arguments, such as NIL, and closed λ-expressions. Note that if $w \Rightarrow v$ then v must be an environment value. Intuitively, the "type" inferred for u by flow analysis is the set of values v such that $u \Rightarrow v$. We can now state a precise relationship between the flow rules and the environment evaluator. First we need a preliminary definition. We say that an assertion Φ contains an environment ρ if either Φ contains ρ directly or ρ is contained in a closure in an environment contained (recursively) in Φ. We say that Φ contains a mapping of x to v if Φ contains an environment ρ such that $\rho(x) = v$. We now have four invariants.

$$\text{if } \texttt{EVAL}(e,\ \emptyset)\ \vdash_E\ \texttt{EVAL}(u,\ \sigma) \quad \text{then } u \text{ is a subterm of } e$$

$$\text{if } \texttt{EVAL}(e,\ \emptyset)\ \vdash_E\ \langle u,\ \sigma\rangle \rightarrow \langle w,\ \rho\rangle \ \text{ then } \ \texttt{EVAL}(e)\ \vdash_F\ u \rightarrow w$$

$$\text{if } \texttt{EVAL}(e,\ \emptyset)\ \vdash_E\ \langle u,\ \sigma\rangle \rightarrow v \ \text{ then } \ \texttt{EVAL}(e)\ \vdash_{F'}\ u \Rightarrow v$$

$$\text{if } \texttt{EVAL}(e,\ \emptyset)\ \vdash_E\ \Phi, \text{ and } \Phi \text{ involves a binding of } x \text{ to } v, \text{ then } \texttt{EVAL}(e)\ \vdash_{F'}\ x \Rightarrow v$$

These conditions can be proved as invariants of the assertions derived by the environment evaluator. This is equivalent to proving them by simultaneous induction on the length of the derivation of $\texttt{EVAL}(e,\ \emptyset)\ \vdash_E\ \Phi$. The proof involves checking that each inference rule in figure 1 preserves all of the above four invariants. We omit this case analysis here. We do, however, note that these four conditions are enough to ensure that if $\texttt{EVAL}(e,\ \emptyset)\ \vdash_E$ ERROR then $\texttt{EVAL}(e)\ \vdash_F$ ERROR. So if flow analysis does not derive ERROR then the input program is safe.

3 Computing Deductive Closures

The flow rules in figure 2 can be viewed as a "bottom-up" logic program. The database community has shown considerable interest in finding efficient strategies for computing bottom-up closures [24]. Here we prove a theorem which allows one to calculate the order of run time of a set of bottom-up rules. Heintze has shown that his version of flow analysis can be computed in $O(n^3)$ time for monomorphic terms [12]. This result is proved here as a corollary of a more general theorem.

By "rule" we simply mean a first order Horn clause. We will represent a rule with the notation $A_1, \ldots, A_n/C$ where A_i and C are first order atomic formulas. We use this notation rather than the more traditional $C : -A_1, \ldots, A_n$ because the nonstandard form is closer to that used in figure 2 and is closer to other representations of type inference rules. The rules in figure 2 can be easily translated into first order Horn clauses.

I will take the term "database" to mean an arbitrary set of ground atomic formulas. Given a set R of first order Horn clauses, and an initial database D, the closure of D under R, denoted $C_R(D)$, is defined to be the set of all ground atomic formulas derivable from D using rules in R. We say that a set of rules R is *terminating* if for every data base D we have that $C_R(D)$ is finite. Since we are interested in rule sets that define bottom-up algorithms, we are primarily interested in terminating rule sets, although the results of this section hold for nonterminating rule sets as well.

We will say that data base D is closed under R if $C_R(D) \subseteq D$. It would seem that determining closedness would be simpler than computing the closure in cases where we are not yet closed. The main theorem of this section states, in essence, that the closure can be computed in the amount of time needed to simple check its closedness. This directly contradicts the intuition that closedness checking is easier than computing the closure — in fact they are equally difficult. This is more a statement about the difficulty of computing the closure than the difficulty of checking closedness — the closure can be computed quickly. The following definition gives a set whose cardinality governs the time needed to check closedness.

Definition: We define a *prefix firing* of a rule $A_1, \ldots, A_n/C$ in a rule set R under data base D to be a ground instance $B_1, \ldots, B_i$ of an initial sequence $A_1, \ldots, A_i$, $i \leq n$, such that $B_1, \ldots, B_n$ are all contained in D. We let $P_R(D)$ be the set of all prefix firings of rules in R for data base D.

To check closedness we can run all the inference rules on the database and see if any new assertions are derived. Since we are thinking of the rule set R as defining an algorithm which takes a database as input, we are interested in the running time as a function of D for fixed R — the so-called "data complexity" of the rule set R. We assume that all terms and formulas in D are given as directed acyclic graphs (dags) so that equality checking can be done in unit time. Using an appropriate indexing scheme one can check whether D is closed under R in $O(|D|+|P_R(D)|)$ time (where D is given in dag form and we are measuring data complexity for a fixed R).

We are primarily concerned here with rules which are written with the intention of defining bottom-up algorithms. In bottom-up interpretation, variables are bound when they occur in antecedents. We will call a variable in the conclusion that does not appear in any antecedent an "unbound variable" of the rule. Although it is possible to define an interpreter which correctly handles unbound variables in conclusions, when writing bottom-up logic programs (sets of inference rules) it is always possible to avoid such variables. A rule without unbound variables in the conclusion will be called *bottom-up bound.* The following theorem states that for antecedent bound rules the closure can always be computed in an amount of time proportional to the "closedness check time".

> **Theorem**: For any set R of bottom-up bound inference rules there exists an algorithm for computing $C_R(D)$ such that if we assume that hash table operations take constant time and that D is given in dag form, the time required to compute $C_R(D)$ is $O(|D|+|P_R(C_R(D))|)$.

For example, let R consist of the rule $x < y,\ y < z,\ P(x,\ y)/\ x < z$. Let D be a set of ground assertions of the form $u < v$ and $P(u,\ v)$. Let n be $|D|$. If $u < v$ occurs in the closure $C_R(D)$ then u and v must both have appeared as top level arguments in D. So there can be only $O(n)$ terms appearing as top level arguments in $C_R(D)$. So $C_R(D)$ can contain at most $O(n^2)$ assertions. However, there can be $O(n^3)$ prefix firings of the first two antecedents, i.e., prefix firings of the form $u < v,\ v < s$. On the other hand there can be only $O(n^2)$ firings of all three antecedents. To see this note that there are only $O(n)$ instances of $P(x,\ y)$ in the closure $C_R(D)$ because assertions of this form are never derived and must come directly from D. For each of the $O(n)$ instances of $P(x,\ y)$ there are only $O(n)$ possible values of the variable z. Since the first two antecedents have $O(n^3)$ firings the above theorems yields an $O(n^3)$ algorithm for computing the closure. The theorem does not, of course, rule out the possibility of a more efficient algorithm. If the rule is rewritten as $P(x,\ y),\ x < y,\ y < z/\ x < z$ then there are only $O(n^2)$ prefix firings and the theorem yields an $O(n^2)$ algorithm.

Now consider the flow rules in figure 2. The rule `EVAL` can be implemented with a set of rules that propagate assertions of the form `EVAL`(e) down the term. Each such rule has at most a linear number of firings. The rules `LAM` and `CONS` also have at most a linear number of firings. All other rules, except `TRANS2`, have at most a quadratic number of firings. The rule `TRANS2` can have a cubic number of firings so, by the above theorem, the time required for the analysis is $O(n^3)$.

To prove the above theorem for bottom-up run time we consider the following source to source transformations on bottom-up logic programs.

$$A_1,\ A_2,\ A_3,\ \ldots A_n/\ C \quad \Rightarrow$$

$$A_1,\ A_2/P(x_1,\ \ldots,\ x_n), \quad P(x_1,\ \ldots,\ x_n),\ A_3,\ \ldots,\ A_n/C$$

where $x_1,\ \ldots, x_n$ are all free variables in A_1 and A_2.

$$A_1, \ldots, P(t_1, \ldots, t_n), \ldots, A_n/C \quad \Rightarrow$$

$$P(t_1, \ldots, t_n)/Q(x_1, \ldots, x_m), \quad A_1, \ldots, Q(x_1, \ldots, x_m), \ldots, A_n/C$$

where at least one of t_i is a non-variable and $x_1, \ldots, x_m$ are all the free variables in $t_1, \ldots, t_n$.

$$P(x_1, \ldots, x_n),\ Q(y_1, \ldots, y_m)/C \quad \Rightarrow$$

$$P(x_1, \ldots, x_n)/P'(f(z_1, \ldots, z_k),\ g(w_1, \ldots w_h))$$

$$Q(y_1, \ldots, y_n)/Q'(g(w_1, \ldots, w_h),\ u(v_1, \ldots v_j))$$

$$P'(x, y),\ Q'(y, z)/R(x, y, z)$$

$$R(f(x_1, \ldots, x_n),\ g(w_1, \ldots, w_h),\ u(v_1, \ldots, v_m))/C$$

where $z_1, \ldots, z_k$ are those variables among the x_is which are not among the y_is; $w_1, \ldots, w_h$ are those variables that occur both in the x_is and y_is; and $v_1, \ldots v_i$ are those variables among the y_is that are not among the x_is.

These transformations yield rules where the only multiple antecedent rules are of the form $P(x,\ y), Q(y,\ z)/R(x,\ y,\ z)$. The theorem is not difficult to prove for such simple rule sets. Furthermore, it can be shown that the transformation leads to only a linear growth in $|P_R(C_R(D))|$.

4 Recursive Types

Flow analysis can be viewed as constructing types — for each subterm u that is a subterm of the input term we have a type defined by the set of values v such that $\texttt{EVAL}(e) \ \vdash_{F'} \ u \Rightarrow v$. But it seems desirable to be able to assign a more traditional type to each term, i.e., to infer a finite *type expression* for each subterm of the input. This can be done by considering type expressions defined by the following grammar.

$$\tau ::= \alpha \mid \bot \mid \tau_1 \rightarrow \tau_2 \mid c(\tau_1, \ldots, \tau_n) \mid \tau_1 \cup \tau_2 \mid \mu\alpha.\tau$$

In the above grammar the symbol $\rightarrow$ is being used to denote functions spaces, i.e., the type $\tau_1 \rightarrow \tau_2$ is the set of functions f such that when f is applied to an element of τ_1 it yields an element of τ_2. We will rely on context to distinguish between uses of $\rightarrow$ to denote function spaces and uses of $\rightarrow$ to denote rewrite or flow arcs. The type represented by the nonterminal Z in the grammar given in section 2 can be written as follows.

$$\mu Z.\texttt{NIL} \cup \texttt{CONS}(a \cup b,\ Z)$$

The type of pure λ-terms can be written as $\mu Z.Z \rightarrow Z$.

The arcs generated by flow analysis can be used to assign a type expression to each of the subterms of the input. We first introduce a type variable α_u for each subterm u of e. We then give these type variables the following mutually recursive "definitions".

$$\alpha_u \equiv \bigcup_{u \to \lambda x.v} \alpha_x \to \alpha_v \qquad \cup \qquad \bigcup_{u \to c(v_1,\ \ldots,\ v_n)} c(\alpha_{v_1},\ \ldots,\ \alpha_{v_n})$$

The unions are taken over flow assertions derived by the rules, e.g., the first union is over all derived flow arcs of the form $u \to \lambda x.v$. If u is subterm of e which does not flow to any constructor applications or λ-expressions then, intuitively, the type of u is taken to be the empty type. In this case the union in the definition of α_u above is empty and by convention the definition becomes $\alpha_u = \bot$. This is the reason for including $\bot$ in type expressions.

The mutually recursive definitions of the type variables α_u can be converted to type expressions with nested recursions. For example, consider the following mutually recursive definitions.

$$\alpha = \mathtt{NIL} \cup \mathtt{CONS}(\alpha,\ \alpha) \cup \mathtt{IF}(\beta,\ \alpha,\ \alpha)$$

$$\beta = \mathtt{EQ}(\alpha,\ \alpha) \cup \mathtt{AND}(\beta,\ \beta)$$

The above mutually recursive definitions can be viewed as a grammar for a simple language of expressions built with the data constructors NIL, CONS, IF, EQ, and AND. The type α can be written as the following *closed* type expression.

$$\mu\alpha.\ \mathtt{NIL} \cup \mathtt{CONS}(\alpha,\ \alpha) \cup \mathtt{IF}(\mu\beta.\mathtt{EQ}(\alpha,\ \alpha) \cup \mathtt{AND}(\beta,\ \beta),\ \alpha,\ \alpha)$$

In general, each type variable occurring in a system of mutually recursive definitions can be expressed with a single closed type expression.

Definition: We write $\mathtt{EVAL}(e)\ \vdash_F\ u : \tau$, $\mathtt{EVAL}(e)\ \not\vdash_F$ ERROR and τ is the type for u constructed from the flow analysis of e in the manner described above.

For example, we have the following.

$$\mathtt{EVAL}(\lambda x.x)\ \vdash_F\ \lambda x.x : \bot \to \bot$$

$$\mathtt{EVAL}(((\lambda x.x)\ a))\ \vdash_F\ \lambda x.x : a \to a$$

Clearly, flow analysis does not construct principal types. Aiken, Wimmers and Lakshman express principal types with constrained types [2, 26] but a discussion of constrained types is beyond the scope of this paper.

Flow analysis type inference is sound, i.e., there exists a natural semantics for type assertions of the form $u : \tau$ such that for any closed term e, if $\mathtt{EVAL}(e)\ \vdash_F\ e : \tau$ then the meaning of e is in the set denoted by τ. To prove this, or even to state it precisely, we need to construct an appropriate denotational semantics for terms and recursive types. The semantics of recursive type expressions presents a challenge. The types allow non-monotone recursion so the Tarski-Knaster fixed-point theorem does not apply. This difficulty is avoided in the ideal semantics of recursive types [19]. Here we present a formulation of the semantics which avoids the use of contractive maps. We adopt a Scott-Strachey denotational semantics in which semantic values are taken to be limits of increasing sequences of finite approximations, analogous to Cauchy sequences of rationals. Let τ be a recursive type expression. We can think of τ as a predicate on finite approximations. The truth of the predicate τ applied to the finite approximation v can be defined by induction on the (finite) rank of v. We can then write $v \in \tau$ if τ is true of v. If s is an infinite sequence of finite approximations we write $s \in \tau$ if $s_i \in \tau$ for all i where s_i is the ith finite approximation in the sequence i. So each type expression τ can be associated with a set of values — the values s such that $s \in \tau$ as just defined. Section 5 contains a rigorous presentation of a version of Scott-Strachey semantics — we define the semantic domain and the denotational semantics of terms. Section 6 rigorously defines the denotational semantics of recursive types and gives a soundness theorem for flow analysis type inference. The denotational semantics defined in section 5 is also used in section 7 to define the typability in the Aiken-Wimmers-Lakshman semantic type system.

5 A Denotational Semantics

In order to prove an equivalence between flow typability and Aiken-Wimmers-Lakshman typability it is necessary to adopt the same semantic domain as that used in [1]. We assume a countably infinite set of data constructors of each arity. We then define an infinite sequence of ordered domains $D_0, D_1, D_2, \ldots$ where D_0 is $\{\bot\}$ and

$$D_{i+1} = D_i \cup (D_i \to_m D_i) \cup \bigcup_c c(D_i - \{\bot\}, \ldots, D_i - \{\bot\}) \cup \{\mathtt{ERROR}\}.$$

Each domain is partially ordered by the least order $\leq$ such that $\bot$ is less than all other elements; a function f is less than a function g if for all x in the domain of f, $f(x) \leq g(x)$; and $c(x_1, \ldots, x_n) \leq c(y_1, \ldots y_n)$ if $x_i \leq y_i$. The expression $D_i \to_m D_i$ represents the set of strict monotone functions from D_i to D_i, i.e., the function f such that $f(\bot)$ is $\bot$ and $x \leq y$ implies $f(x) \leq f(y)$. Since D_i is finite all functions from D_i to D_i are continuous. We take D_ω to be the union of all D_i and we assign a partial order to D_ω in the same way. D_ω is the set of all finite approximations.

We now define a D_ω-chain to be an infinite sequence s of elements of D_ω such that $s_i \leq s_{i+1}$ for all i where s_i is the ith element of the chain. We use D_ω-chains to represent semantic values. Representing semantic values with D_ω-chains is analogous to representing real numbers with Cauchy sequences of rationals. Note that if a chain element s_j is of the form $c(v_1, \ldots, v_n)$ then there must exist chains $w^1, \ldots, w^n$ such that for $i > j$ we have s_i equals $c(w_i^1, \ldots, w_i^n)$. More generally we have that every chain can be uniquely classified as either $\bot$ (the constant chain containing only $\bot$) or ERROR (any chain containing some number of occurrences of $\bot$ followed by an infinite chain of the value ERROR), a constructor applied to chains, or a chain of functions, i.e., a function chain. Given two elements v_1 and v_2 of D_ω we define $(v_1\ v_2)$ to be ERROR if v_1 is a constructor application or either v_1 or v_2 are ERROR, $\bot$ if v_i is either $\bot$ or a function not containing v_2 in its domain, and $v_1(v_2)$ otherwise. For chains s and w we define $(s\ w)$ to be the chain u such that u_i is $(s_i\ w_i)$. Similarly, for any D_ω-chains $s^1, \ldots, s^n$ we define $c(s^1, \ldots, s^n)$ to be the D_ω chain w where w_i is $\bot$ if any s_i^k is $\bot$, ERROR if some s_i^k is ERROR, and $c(s_i^1, \ldots, s_i^n)$ otherwise. For $v \in D_\omega$ and s a D_ω-chain we write $v \leq s$ if there is some s_i such that $v \leq s_i$. For two D_ω-chains s and u we write $s \leq u$ if $s_i \leq u$ for all i. We write $s \equiv u$ if $s \leq u$ and $u \leq s$. For each i and D_ω-chain s there is a greatest $v \in D_i$ such that $v \leq s$. For any D_ω-chain s we let $|s|$ be the D_ω-chain such that $|s|_i$ is the greatest v in D_i such that $v \leq s$. We have that $|s|$ is a normal form of s, i.e., we have that $s \equiv w$ if and only if $|s| = |w|$.

A denotational environment is a mapping from term variables to D_ω-chains. For any denotational environment σ and term e the meaning of e under σ, denoted $[\![e]\!]\sigma$, is defined below. The notation $([\![e]\!]\sigma)_i$ denotes the ith element of the chain $[\![e]\!]\sigma$. The notation $\sigma[x := v]$ for $v \in D_\omega$ denotes the variable interpretation which is identical to σ except that it maps x to the constant chain v, i.e., the chain s such that $s_i = v$ for all i.

$$[\![x]\!]\sigma = \sigma(x)$$

$$[\![(e_1\ e_2)]\!]\sigma = ([\![e_1]\!]\sigma\ [\![e_2]\!]\sigma)$$

$$([\![\lambda x.u]\!]\sigma)_i = \left\{\langle v_1,\ v_2\rangle :\ v_1 \in D_i \wedge v_2 = \left\{\begin{array}{ll}\bot & \text{if } v_1 = \bot \\ |[\![u]\!]\sigma[x := v_1]|_i & \text{otherwise}\end{array}\right\}\right\}$$

$$[\![c(u_1,\ \ldots,\ u_n)]\!]\sigma = c([\![u_1]\!]\sigma,\ \ldots,\ [\![u_n]\!]\sigma)$$

$$[\![\texttt{CASE}(e \text{ of } c(x_1,\ \ldots,\ x_n){:}\,u,\ y{:}\,w)]\!]\sigma = \begin{cases}\bot & \text{if } [\![e]\!]\sigma = \bot \\ \text{ERROR} & \text{if } [\![e]\!]\sigma = \text{ERROR} \\ [\![u]\!]\sigma[x_1 := s_1,\ \ldots,\ x_n := s_n] & \text{if} [\![e]\!]\sigma = c(s_1,\ \ldots,\ s_n) \\ [\![w]\!]\sigma[y := [\![e]\!]\sigma] & \texttt{otherwise}\end{cases}$$

We now have a denotational domain and a denotational semantics for terms.

6 The Denotational Semantics of Recursive Types

Here we construct a denotational semantics for type expressions in the setting of the denotational domain defined in section 5. Semantic types are represented by sets of finite approximations, i.e., subsets of D_ω. If S is a subset of D_ω and s is a D_ω-sequence, then by abuse of notation we write $s \in S$ if $s_i \in S$ for all i. We can now define the meaning of a recursive type expression as a subset of D_ω. First, we convert type expressions to expressions in a certain normal form. For example, the normal form of $\mu\alpha.\alpha \cup \tau[\alpha]$ is $\mu\alpha.\bot \cup \tau[\alpha]$. More generally, in any recursive type a recursive call that does not occur inside a function space constructor or a data constructor is replaced by $\bot$. The recursive types denote least fixed-points. Greatest fixed-points could be achieved by replacing top level recursions by a universal type. Normal forms, in which all recursions occur inside function spaces or constructors, have exactly one fixed-point so there is no need to distinguish least from greatest.

We now give a recursive definition of an element relation $\in$ between values v in D_ω and closed recursive type expressions. Recall that all elements of D_ω are finite. The definition of the element relation $\in$ is defined by the following cases.

1. $v \in \bot$ if $v = \bot$.
2. $v \in \tau_1 \to \tau_2$ if v is either $\bot$ or a (finite) function such that for all x in the domain of v, if $x \in \tau_1$ then $f(x) \in \tau_2$.
3. $v \in c(\tau_1, \ldots, \tau_n)$ if either v is $\bot$ or v is of the form $c(x_1, \ldots, x_n)$ and for each x_i we have $x_i \in \tau_i$.
4. $v \in \tau_1 \cup \tau_2$ if $v \in \tau_1$ or $v \in \tau_2$.
5. $v \in \mu\alpha\tau[\alpha]$ if $v \in \tau[\mu\alpha.\tau[\alpha]]$

Since we are only concerned with closed type expressions we do not need to consider type variables in the above definition. Furthermore, if τ is in normal form, i.e., all recursion occurs inside function spaces or data constructors, and v is an element of D_ω, then the above recursion terminates because any given element of D_ω is finite and the recursion eventually reduces the size (rank) of the finite approximation. The recursive definition of $\in$ has exactly one fixed-point because recursive calls reduce a well founded order (for elements of D_ω and normal form type expressions). If s is a D_ω chain then we write $s \in \tau$ if $s_i \in \tau$ for all i. We write $\tau_1 \leq \tau_2$ if $v \in \tau_1$ implies $v \in \tau_2$ and we write $\tau_1 \equiv \tau_2$ if $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$.

We now define a type context to be a set of assertions of the form $x : \tau$ where x is a term variable and τ is a closed recursive type expression. If σ is a denotational environment, i.e., a mapping from term variables to D_ω-sequence, and Σ is a type context, then we write $\sigma \models \Sigma$ if for each assertion $x : \tau$ in Σ we have $\sigma(x) \in \tau$. For any type environment Σ, term u, and closed type τ, we write $\Sigma \models u : \tau$ if for all denotational environments σ such that $\sigma \models \Sigma$ we have that $[\![u]\!]\sigma \in \tau$.

The flow analysis of a term assigns a type to all the variables in that term, including all bound variables in the term. In a flow derivation of the form $\mathtt{EVAL}(e) \;\vdash_f\; u : \tau$ we can think of e as providing a context in which to interpret the free variables of u. More formally, for any term e we define Σ_e to be the set of assertions of the form $x : \gamma$ where x is a variable occurring in e and $\mathtt{EVAL}(e) \;\vdash_F\; x : \gamma$. We now have the following soundness theorem for flow type inference.

Theorem: If $\mathtt{EVAL}(e) \;\vdash_F\; u : \tau$, then $\Sigma_e \models u : \tau$.

For example, let e be the term $((\lambda h.(h\ h))\ \lambda x.x)$. Flow analysis yields $h \to \lambda x.x$ and $x \to \lambda x.x$. This leads to the type definitions $\alpha_h = \alpha_x \to \alpha_x$ and

$\alpha_x = \alpha_x \to \alpha_x$. We now have $\texttt{EVAL}(e) \ \vdash_F \ h : \mu\alpha.\alpha \to \alpha$. The above theorem now implies the following.

$$h : \mu\alpha.\alpha \to \alpha \models (hh) : \mu\alpha.\alpha \to \alpha$$

Recall that by definition $\texttt{EVAL}(e) \ \vdash_F \ u : \tau$ implies that $\texttt{EVAL}(e) \not\vdash_F \texttt{ERROR}$ so the above theorem only applies when flow analysis fails to derive ERROR. It is easy to see that sets denoted by the recursive type expressions do not contain ERROR. Hence if e is a closed term and τ is a recursive type such that $\models e : \tau$ then the value of e can not be ERROR, i.e., e is semantically safe. The above theorem now implies that if $\texttt{EVAL}(e) \not\vdash_F \texttt{ERROR}$ then $\models e : \tau$ for some recursive type τ and hence e is semantically safe. So the above theorem provides an alternative proof that if flow analysis fails to derive ERROR then the input term is safe — intuitively, failure to derive ERROR implies typability by recursive types.

The proof of the above theorem is directly analogous to a soundness proof for the Aiken-Wimmers-Lakshman type inference rules. Here we proceed by structural induction on the term u. For any subterm u of e we let τ_u be the type constructed for u, i.e., the type such that $\texttt{EVAL}(e) \ \vdash_F \ u : \tau_u$. The case of variables is trivial — if $\texttt{EVAL}(e) \ \vdash_F \ x : \tau$ then $x : \tau$ is an element of Σ_e so $\Sigma_e \models x : \tau$. Now suppose u is $\lambda x.w$. Let $\Sigma_{e,\lambda x.w}$ be the subset of Σ_e which assigns types to free variables of $\lambda x.w$. By the induction hypothesis we have $\Sigma_{e,\lambda x.w}; x : \tau_x \models w : \tau_w$. This implies $\Sigma_{e,\lambda x.w} \models \lambda x.w : (\tau_x \to \tau_w)$. The most difficult case is application. Consider an application $(f\ w)$. We will show that $\tau_f \subseteq \tau_w \to \tau_{(f\ w)}$. This suffices since by the induction hypothesis we then have $\Sigma_e \models f : (\tau_w \to \tau_{(f\ w)})$ and $\Sigma_e \models w : \tau_w$. To show $\tau_f \subseteq \tau_w \to \tau_{(f\ w)}$ we first note that since $\texttt{EVAL}(e) \not\vdash_F \texttt{ERROR}$ we must have that τ_f is either $\bot$ or a union of function types of the form $\tau_x \to \tau_m$ where $\texttt{EVAL}(e) \ \vdash_F \ f \to \lambda x.m$. Now there are two subcases. First suppose that $\texttt{EVAL}(e) \not\vdash_F \exists w$. It is possible to show that in this case τ_w must denote the set $\{\bot\}$. But in a strict semantics we have that $\bot \to \gamma$ contains all function types so we are done. Now suppose that $\texttt{EVAL}(e) \not\vdash_F \exists w$. In this case, for each λ-expression $\lambda x.m$ such that $\texttt{EVAL}(e) \ \vdash_F \ f \to \lambda x.m$ we have $\texttt{EVAL}(e) \ \vdash_F \ x \to w$ and $\texttt{EVAL}(e) \ \vdash_F \ (f\ w) \to m$. But this implies that $\tau_w \subseteq \tau_x$ and $\tau_m \subseteq \tau_{(f\ w)}$. So $\tau_x \to \tau_m \subseteq \tau_w \to \tau_{(f\ w)}$. So we get $\tau_f \subseteq \tau_w \to \tau_{(f\ w)}$ as desired and the application case is complete. Constructors and case statement are handled analogously by showing that in each case the preconditions of the appropriate Aiken-Wimmers-Lakshman type inference rules hold.

7 Typability by Semantic Types

Aiken, Wimmers and Lakshman (AWL) define a semantic type system. As in section 4, types are taken to be certain subsets (ideals) of a semantic domain [1, 2, 19]. They define type inference rules in which type variables range over semantic types (sets of values). They show that even with this very liberal notion of type, typability of a given term is still decidable and typability implies operational safety. We prove here that for monomorphic programs with shallow case statements typability in the AWL system is equivalent to typability by the flow analysis defined in section 2. Typability of polymorphic terms (terms with let) is equivalent to typability of the let-expansion of the term. So typability by flow analysis also provides a characterization of the typable polymorphic terms. Polymorphism is discussed in more detail in section 8. Typability of terms involving deep case statements can also be handled by generalizations of flow analysis techniques but this is beyond the scope of this paper.

We again consider the semantic domain D_ω defined in section 5. Here we take a type to be an arbitrary subset of D_ω. The type inference system of Aiken, Wimmers and Lakshman can be expressed using some additional notation for expressing types. If τ and σ are types we let $\tau \to \sigma$ be the set of elements of D_ω which are either $\perp$ or are (finite) functions mapping elements of τ to elements of σ. For any type τ we define $\tau|_c$ to be the subtype of τ consisting of $\perp$ plus the elements of τ corresponding to applications of the constructor c. We let $\tau - \tau|_c$ be the subtype of τ consisting of values other than applications of c. If τ and σ are types we let $\tau?\sigma$ be the type τ if σ contains an element other than $\perp$ and $\{\perp\}$ otherwise. We also let $c(\sigma_1, \ldots, \sigma_n)$ be the type containing $\perp$ and all values of the form $c(v_1, \ldots, v_n)$ where v_i is an element of σ_n other than $\perp$. We also write $\tau \cup \sigma$ to denote standard set-theoretic union.

Aiken, Wimmers and Lakshman present their system as type inference rules. However, for monomorphic programs in which distinct variables have distinct names, a term is typable in their system if and only if each subterm can be assigned a type such that the system of type assignments satisfies certain constraints. For terms u and e we write $u \in e$ to indicate that u is a subterm of e. For any monomorphic term e we define $\mathtt{AWL}(e)$ to be the following system of constraints on semantic types τ_u for $u \in e$.

$$\text{if } \lambda x.u \in e \text{ then } \tau_{\lambda x.u} = \tau_x \to \tau_u$$

$$\text{if } (f\ u) \in e \text{ then } \tau_f \subseteq (\tau_u \to \tau_{(f\ u)})$$

$$\text{if } c(u_1, \ldots, u_n) \in e \text{ then } \tau_{c(u_1, \ldots, u_n)} = c(\tau_{u_1}, \ldots, \tau_{u_n})$$

$$\text{if } w \equiv \mathtt{CASE}(e_1 \text{ of } c(x_1, \ldots, x_n){:}e_2,\ y{:}e_3) \in e \text{ then}$$

$$\tau_{c(x_1, \ldots, x_n)} = \tau_{e_1}|_c \quad \tau_y = (\tau_{e_1} - \tau_{e_1}|_c) \quad \tau_w = \tau_{e_2}?(\tau_{e_1}|_c) \cup \tau_{e_3}?\tau_y$$

Definition: A monomorphic term e is AWL typable if the constraint set $\mathtt{AWL}(e)$ is solvable over semantic types.

We now have the following theorem.

Theorem: A monomorphic term e is AWL typable if and only if $\mathtt{EVAL}(e)\ \not\vdash_F\ \mathtt{ERROR}$.

First we show that if $\mathtt{EVAL}(e)\ \not\vdash_F\ \mathtt{ERROR}$ then e is AWL typable. Consider a fixed term e and for each subterm u of e define τ_u to be the type expression such that $\mathtt{EVAL}(e)\ \vdash_F\ u : \tau_u$. We need to show that the sets denoted by τ_u satisfy the constraint set $\mathtt{AWL}(e)$. But this is exactly what was shown in the proof of the soundness theorem in section 6.

We now show that if e is AWL typable then EVAL(e) $\not\vdash_F$ ERROR. Consider a fixed term e. For technical reasons we first argue that the constraints are solvable if and only if there is a solution in which no type τ_u is the universal type (the entire set of safe values). Let c be a constructor constant not occurring in e. Consider the semantic domain derived by removing c from the set of the constructor constants. This new semantic domain is isomorphic to the original because the number of constructor constants is a-priori infinite. Hence if there is a solution over the original domain then there is also a solution over the new domain. But the new domain is a subset of the original and the solution over the new domain is a solution over the original in which no type is universal. Hence we can assume that no type is universal.

Now we show that if the constraints are solvable by non-universal types then the rules do not derive ERROR. The basic idea is to show that the flow rules in figure 2 are sound in the sense that they derive information that is true in *any* solution to AWL(e) where no type is universal. Consider a solution to the constraint set AWL(e) in which no type is universal. We now let τ_u be the semantic type assigned to u in the solution to the constraint set AWL(e). By induction on the inference process we can show that if the rules derive $\exists\ e$ then τ_e contains some value other than $\perp$ and if the rules derive the flow arc $e \to u$ then $\tau_u \subseteq \tau_e$. These invariants ensure that the rule BETA3 can never fire because the constraint set AWL(e) ensures that τ_{e_1} does not contain constructor values.

To prove that the inference rules preserve the invariants we must check each rule — ensuring that the desired properties of the conclusion of the rule follow from the assumed properties about the antecedents of the rule. For the most part this is a straightforward "soundness analysis" of the inference rules. The most subtle rule is BETA2. From the induction hypothesis and the antecedents of the rule we have $\tau_{\lambda x.u} \subseteq \tau_{e_1}$. From the above constraints we have $\tau_{\lambda x.u} = \tau_x \to \tau_u$ and $\tau_{e_1} \subseteq \tau_{e_2} \to \tau_{(e_1\ e_2)}$. Therefore we have the following.

$$\tau_x \to \tau_u \subseteq \tau_{e_2} \to \tau_{(e_1\ e_2)}$$

Note that under strict semantics if either τ_{e_2} is $\perp$, or $\tau_{(e_1\ e_2)}$ is universal, then $\tau_{e_1} \to \tau_{(e_1\ e_2)}$ is the set of all functions and nothing follows from the above inclusion. However, from the antecedents of the rule and the induction hypothesis we have that τ_{e_2} is nonempty. We have assumed that $\tau_{(e_1\ e_2)}$ is not universal. So we now have that $\tau_{e_2} \subseteq \tau_x$ and and $\tau_u \subseteq \tau_{(e_1\ e_2)}$. This establishes the conditions needed to prove the invariants for the conclusions of the rule. We omit the soundness analysis of the other inference rules.

8 Let-Polymorphism

So far we have only considered monomorphic programs, i.e., those not containing let-expressions. We can specify an appropriate notion of typability for polymorphic terms considering let-expansions. The let-expansion of a term is the result of repeatedly applying the rewrite rule (LET x = e in u) $\to u[e/x]$ until the term does not contain any more let-expressions. Every expression has a unique let-expansion, although that expansion may be exponentially larger than the original. More efficient approaches to let-polymorphism are discussed in section 9. For example consider (LET $I = \lambda x.x$ in $(\lambda xy.x\ (I\ a)\ (I\ b))$). The let-expansion of this term is $(\lambda xy.x\ ((\lambda x.x)\ a)\ ((\lambda x.x)\ b))$. This can be α-renamed to $(\lambda xy.x\ ((\lambda w.w)\ a)\ ((\lambda z.z)\ b))$. Flow analysis now gives only one value of w, and one value of the overall term. Intuitively, the different occurrences of the identity function are assigned different types. Note that α-renaming is important to avoid confusing different bound variables of the same name.

> **Definition:** A polymorphic term e is *flow typable* if there exists an α-variant e' of the let-expansion of e such that $\mathtt{EVAL}(e') \not\vdash_F \mathtt{ERROR}$.

Since let-expansions can be done in exponential time flow typability is exponential time decidable. Even for simple types, the problem of determining polymorphic typability (typability of the let-expansion) is complete for exponential time [20, 18]. So one could argue that monomorphic analysis of the let-expansion is the best that can be done. However, we show in section 9 that for any fixed bound on order and arity, the Hindley-Milner type inference algorithm for determining polymorphic typability can be made to run in pseudo-linear time, i.e., $O(n\alpha(n))$ where α is the inverse Ackerman function. So in practice Hindley-Damas typability can be determined without incurring the cost of let-expansion. Aiken, Wimmers and Lakshman have constructed a type inference algorithm analogous to the Hindley-Milner algorithm [2]. The algorithm works with "constrained types" of the form $\forall \boldsymbol{\alpha}\!:\!\Phi\ \tau$ where τ is a type expression, $\boldsymbol{\alpha}$ is a sequence of type variables, and Φ is a set of constraints that must be satisfied by $\boldsymbol{\alpha}$. Constrained types generalize the type schemes of traditional Hindley-Milner type inference. Constrained types also provide a means of expressing principal (or least) types. Although the Aiken-Wimmers-Lakshman generalization of the Hindley-Milner procedure may prove effective in practice, we show here that the pseudo-linear time result for bounded order and arity fails for polymorphic flow typability.

> **Theorem:** Polymorphic flow typability is PSPACE hard even for first order terms of arity one.

This theorem leaves a gap in the complexity of polymorphic flow typability for order one and arity one — we have that it is decidable in exponential time by let-expansion and that it is PSPACE hard.

The proof is by reduction of the problem of determining whether a given Turing machine accepts a given input when computation is limited to a tape of length n. This problem is trivially PSPACE complete. For each symbol in the tape alphabet we introduce a unary constructor c so that terms of the form $c_1(c_2(\ldots c_n(a)))$ represents a state of the tape in the obvious way. For each state of the machine we have a unary constructor s distinct from the tape alphabet constructors. A complete state of the computation can be described by a term of the form $c_1(c_2(\ldots c_i(s(c_{i+1} \ldots c_n(a)))$ with a single state symbol occurring at the position just after the symbol being currently read by the head of the Turing machine. The transition table of the Turing machine can be represented by a large, but still polynomial, function f constructed from case statements and having the property that $f(c_1(s(c_2(x))$ computes to a term of the form $s'(c'(c_2(x))$ or $c'(c_2(s'(x))$ depending on whether the head moves left or right respectively when reading tape symbol c_1. For a tape of length n we construct n different versions $f_1, \ldots, f_n$ of the transition function such that each version satisfies the above condition, and furthermore $f_i(c_1(c_2(x))$ computes to $c_1(f_{i+1}(c_2(x))$. Intuitively, rather than define a single transition function which calls itself recursively when the head symbol is not at the front of the list, we define n copies of the function where f_1 "recurses" to f_2 and so on up to n — the length of the tape. We then have that if M is a term representing a state of the computation then $f_1(M)$ is the next state. By using a non-recursive program with separate copies of the transition function for each position of the tape, the flow rules given an *exact* analysis of $f_1(M)$ — the type of $f_1(M)$ constructed by the flow analysis contains the single term which is the value of $f_1(M)$. We now construct functions $g_1, \ldots, g_k$ where g_1 is f_1 and g_{i+1} is LET $h = g_i$ in$\lambda x.(h(h(x)))$. If M_0 represents the initial state, then $g_i(M_0)$ denotes the state after 2^i steps of computation. We assume that if the machine ever reaches an accepting state then it remains in an accepting state from that point on. If the machine reaches an acceptable state it must do so within an exponential number of transitions since otherwise the computation must have entered a non-accepting loop. We now have that the computation reaches an accepting state if and only if $g_k(M_0)$ is an accepting state where k is at least $n \log |A|$ where $|A|$ is the size of the tape alphabet. Using techniques similar to those used to construct f_1, we can write a function w such that $w(M)$ generates an error if and only if M is not an accepting state and such that flow-analysis is exact for applications of w. Note that we can use the term $\texttt{CASE}(e \texttt{ of } p{:}u,\ y{:}(\texttt{CASE}(y \texttt{ of } q{:}c,\ z{:}c)\ c))$ to generate a flow analysis error exactly when e flows to a value that is not an instance of p. We now have that $w(g_k(M_0))$ is polymorphically flow typable if and only if the given machine reaches an accepting state. Furthermore, (the let-expansion of) $w(g_k(M_0))$ is first order, uses only monadic functions, and the type of every subterm of the let-expansion has size at most n.

9 Polymorphic Analysis in Pseudo-Linear Time

Programmers almost never write functions with more than ten arguments or function of order larger than three (functions whose arguments are functions whose arguments are functions). In practice, programs have bounded arity and order. Bounding arity and order bounds the size of types. Henglein [15] proved that typability by types of bounded size is polynomial time decidable in both the Hindley-Milner and Milner-Mycroft type systems. Henglein's result implies that for bounded order and arity typability is polynomial time decidable. Here we show that for any bound on order and arity type inference can be done in pseudo-linear time, i.e., $O(n\alpha(n))$ where α is the inverse Ackerman function. This can be achieved simply by using a pseudo-linear unification algorithm (based on union-find) in the standard Hindley-Milner type inference procedure. Both the Hindley-Milner type inference procedure and pseudo-linear unification based on union-find are well known. We review them here only for the purpose of analyzing the running time of their joint use. It is interesting to note that standard linear time unification procedures result in quadratic overall performance — linear procedures are batch and here we need good on-line performance. We also show that the standard Hindley-Milner type inference procedure can be made to simulate union-find. This suggests that the Hindley-Milner procedure can not be made to run in truly linear time. It is still possible, however, that there exists some other linear time algorithm for deciding typability.

For Hindley-Milner type inference we consider the language generated by the following grammar.

$$e ::= x \mid 0 \mid (e_1\, e_2) \mid \lambda x.e \mid \mathrm{LET}\ x = e_1\ \mathtt{in}\ e_2$$

We also consider type expressions defined by the following grammar.

$$\tau ::= N \mid \tau_1 \rightarrow \tau_2$$

A term e is Hindley-Milner typable if there exists an α-variant e' of the let-expansion of e such that for each $u \in e'$ there exist a type expression τ_u satisfying the following "unification constraints".

$$\tau_0 = N \quad \tau_{\lambda x.e} = (\tau_x \rightarrow \tau_e)$$
$$\text{if } (f\ u) \in e' \text{ then } \tau_f = (\tau_u \rightarrow \tau_{(f\ u)})$$

Hindley-Milner typability guarantees safety, i.e., that the constant 0 is never applied as a procedure during the evaluation of the term. We can also add constants for arithmetic operations and fixed point operators for constructing recursive functions (the untyped Y operator is not Hindley-Milner typable). Since unifiability is linear-time decidable, Hindley-Milner typability is linear-time decidable for monomorphic terms.

We define the size of an expression e, written $|e|$, (where e is either a term or a type expression) to be the number of subexpression occurrences in e, i.e., the number of positions occupied in e. The size of an expression is proportional to the length of the expression when written out. For example, the size of $(N \rightarrow N) \rightarrow (N \rightarrow N)$ is seven even though only three distinct subexpressions are involved.

Now we define the concepts of arity and order. Every type expression is either N or is of the form $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \tau_n \rightarrow N$ where $\rightarrow$ associates to the right. Intuitively, functions of this type take n arguments and we can write $(f\ e_1\ e_2\ \ \ldots\ e_n)$ as an abbreviation for $(\ldots((f\ e_1)\ e_2)\ldots\ e_n)$. We can also write write $\tau_1 \times \ldots \times \tau_n \rightarrow N$ as a syntactic variant of the type $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \tau_n \rightarrow N$. The arity of N is defined to be zero and the arity of $\tau_1 \times \ldots \times \tau_n \rightarrow N$ is defined to be n. We define the order of N to be zero and the order of $\tau_1 \times \ldots \times \tau_n \rightarrow N$ to be one greater than the largest order of an argument type τ_i. If τ, and all subexpressions of τ, have arity at most a and order at most o then $|\tau|$ is $O(a^o)$. So bounding arity and order also bounds the size of the type.

Definition: A term e is Hindley-Milner-typable by types of size b if there exists an α-variant e' of the let-expansion of e and a solution to the unification constraints of e' such that for each type τ_u in that solution we have $|\tau_u| \leq b$.

Here we prove the following theorem.

Theorem: Hindley-Milner typability of a term e by types of size b can be determined in $O(bn\alpha(bn))$ time where α is the inverse of Ackerman's function and n is the size of e.

The proof is based on a version of the standard Hindley-Milner type inference algorithm using a unification procedure based on union-find. We first note a technicality introduced by the use of let-expansion in the above definition. If a let-bound variable is not used in the body then let-expansion can eliminate ill-formed terms. So the above definition of typability accepts more terms than the traditional definition based on typing rules for let-expressions. However, one can, in linear time assuming constant time table operations, eliminate all let-expressions of the form `LET` $x = e_1$ `IN` e_2 where x does not appear free in e_2. So we can assume that in every let-expression the let-bound variable occurs free in the let-expansion of the body. In this case the standard Hindley-Milner type inference procedure can be used to determine typability as defined above.

Now we review the Hindley-Milner type inference algorithm [21]. We define a *type scheme* to be an expression of the form $\forall \boldsymbol{\alpha} \tau$ where $\boldsymbol{\alpha}$ is a sequence of type variables and τ is a type expression built from N, $\rightarrow$ and type variables. Types are taken to be a special case of type schemes where $\boldsymbol{\alpha}$ is the empty sequence. We let $FTV(\tau)$ denote the free type variables of τ where τ is either a type or type scheme. The algorithm can be formalized as a recursive procedure HM which takes a term e, asserts a system of equations between expressions involving type variables, and returns a type variable representing the "principal type" of e. The input term is typable by types of size b if and only if this system of equations has a solution in which every type variable is interpreted as a type of size b or less. We assume that term variables are renamed so that semantically distinct variables have distinct names. We also assume that for every variable x which is not let-bound we have a type variable α_x. Each let-bound variable x becomes associated with a type scheme $\rho(x)$ which is computed during the analysis of the let-expression. For any type scheme τ we let $Fresh(\tau)$ be a type expression derived by replacing all the bound variables of τ by fresh type variables.

To Compute $HM(e)$:

1. If e is a variable x that is not let-bound then return the type variable α_x.
2. If e is a let-bound variable x then let β be a new type variable; let τ be the type scheme $\rho(x)$; assert the equation $\beta = Fresh(\tau)$; and return β.
3. If e is an application $(e_1\ e_2)$ then let α_{e_1} and α_{e_2} be $HM(e_1)$ and $HM(e_2)$ respectively. Let γ and δ be two fresh type variables. Assert the equations $\alpha_{e_1} = \gamma \rightarrow \delta$ and $\alpha_2 = \gamma$ and return δ.

4. If e is a λ-expression $\lambda x.u$ then let α_u be $HM(u)$ recursively. Let β be a new type variable; assert the equation $\beta = \alpha_x \rightarrow \alpha_u$; and return β.
5. If e is the let-expression LET $x = e_1$ IN e_2 then let α_{e_1} be $HM(e_1)$ recursively. Let τ be the type expression $\sigma(\alpha_{e_1})$ where σ is the most general unifier of the equations constructed to this point (after processing e_1). Set $\rho(x)$ to be the scheme $\forall\boldsymbol{\alpha}\tau$ where $\boldsymbol{\alpha}$ consists of all free type variables α of τ such that there is no free variable x in e_1 such that $\sigma(\alpha) = \sigma(\alpha_x)$. Return $HM(e_2)$.

We assume that equations are processed as they are asserted. The procedure for processing equations uses union-find to maintain an equivalence relation on type expressions. It also maintains a "canonical representative" of each equivalence class where the canonical representative of the class containing τ is stored in $c[find(\tau)]$. We assume that each program variable x is associated with an index such that in any let-expression LET $x = e_1$ IN e_2 we have that the free variables of e_1 have index less than the index of x and bound variables of e_1 have index greater than the index of x. The canonical representative is maintained in such a way that if an equivalence class contains only variables, and at least one member of the class is of the form α_x for a term variable x, then the canonical is α_y where y is the least-indexed such term variable. This will allow step 5 in the above procedure to be changed so that the phrase "variables α of τ such that there is no free variable x in e_1 such that $\sigma(\alpha) = \sigma(\alpha_x)$" to be replaced by "variables α other than those of the form α_y where y has index less than x". The latter test is more efficient and allows the type scheme $\forall\boldsymbol{\alpha}\tau$ to be computed in $O(b)$ time given that τ has size b or less. We now give the procedure for processing equations. In this specification we simply assume that all type variables have indices and that the canonical of an equivalence class containing only variables is the variable of least index.

> **To assert the type equation $\tau_1 = \tau_2$ do the following:** Let f_1 be $find(\tau_1)$, let f_2 be $find(\tau_2)$. If $f_1 = f_2$ do nothing. Else, call $union(\tau_1,\ \tau_2)$ and then let f_3 be $find(\tau_1)$ (f_3 is the find for the new equivalence class). If $c[f_1]$ and $c[f_2]$ are both non-variables and start with different function symbols then fail (a clash has occurred). If both $c[f_1]$ and $c[f_2]$ are non-variables and start with the same function symbol then set $c[f_3]$ equal to $c[f_1]$ and assert equations between the corresponding arguments of $c[f_1]$ and $c[f_2]$. If one of $c[f_1]$ or $c[f_2]$ is a non-variable then set $c[f_3]$ equal to that non-variable and return. If both $c[f_1]$ and $c[f_2]$ are variables then set $c[f_3]$ to be the canonical of smaller index and return.

Let σ be the mgu of the equations prior to asserting $\tau_1 = \tau_2$. If $\sigma(\tau_1)$ and $\sigma(\tau_2)$ both have size at most b then processing the $\tau_1 = \tau_2$ can produce at most $O(b)$ additional assertions of equations. This allows us to bound the processing of equations so that if they produce more than a certain bounded number of recursive invocations the processing simply fails. This "overflow failure" can never occur when all type expressions are no larger than b and ensures that calls to equation processing involve at most $O(b)$ calls to union and find.

This equation processing procedure does not perform occurs-check. We could incorporate an occurs-check into the procedure but it seems simpler to delay the occurs check until the very end of the type inference process as described below. As given, the procedure will performs unification over rational expressions (possibly infinite expressions with only a finite set of distinct subexpressions).

The canonical terms associated with equivalence classes encode the mgu of the processed equations. The following procedure computes the term $\sigma(\tau)$ where σ is the mgu of the equations processed so far.

To compute $\sigma(\tau)$**:** Let v be $c[find(\tau)]$. If v is a variable return it. If v is $f(\tau_1, \ldots, \tau_n)$ return $f(\sigma(\tau_1), \ldots, \sigma(\tau_n))$.

Since the unification procedure does not have an occurs-check test the above substitution procedure might diverge. However, we require that the returned expression have size no larger than b. Hence we can impose an "timeout" on the computation of the substitution so that the applying the substitution either terminates of fails in $O(b)$ time and failure implies failure of the overall type inference process.

It is now not difficult to check that, ignoring the time spent in union-find operations, the procedure takes $O(b)$ time per node of the input term. This implies that the total number of union-find operations is $O(bn)$ where n is the size of the input term. So the procedure terminates in $O(bn\alpha(bn))$ time assuming we use the well known pseudo-linear union-find procedure.

Finally, we must check that for all type expressions τ appearing in the asserted equations we have that $\sigma(\tau)$ has size at most b. This is done by accumulating a set of assigned sizes. We first assert that if $c[find(\tau)]$ is a constant or variable then $\sigma(\tau)$ has size 1. We then close the set of assigned sizes under the principle that if $c[find(\tau)] = f(\tau_1, \ldots, \tau_n)$ and the size of $\sigma(\tau_i)$ is known for each τ_i then $|\sigma(\tau)| = 1 + |\sigma(\tau_1)| + \ldots + |\sigma(\tau_n)|$. This closure can be computed in time linear in the number of type expressions involved which is $O(bn)$. If any such size is greater than b then the overall type inference process fails. If any type expressions fail to have an assigned size after this calculation then there is an occurs-check violation (some type is infinite) and the overall inference process also fails. The overall time taken by the procedure is $O(bn\alpha(bn))$.

The pseudo-linear unification procedure is useful in this application because we need to be ably to apply most general unifiers at various intermediate points of the computation. In other words, we need an on-line unification procedure. Linear time unification procedures are batch procedures and result in quadratic overall performance in this application. However, it is natural to ask whether the power of union-find is really needed and whether a truly linear procedure exists for Hindley-Milner typability by types of bounded size. Here we show that no such procedure is likely to be constructed by simply using a different unification procedure. In particular we show that any sequence of union-find operations can be converted to a term such that the Hindley-Milner type inference algorithm applied to that term computes the values of the find operations in that sequence. Since no truly linear union-find algorithm is known, it seems unlikely that the above algorithm for Hindley-Milner type inference can be made to run in linear time (no matter how unification is implemented).

First we consider a sequence of union operations followed by a single find operation — $union(u_1,\ v_1), \ldots, union(u_n,\ v_n)$ followed by $find(w)$. We let s be the term $\lambda x.\lambda y.y$. We let $u_i = v_i$ be an abbreviation for the term $\lambda f.s((f\ u_i)\ (f\ v_i))$. We let $progn(e_1,\ e_2,\ \ldots,\ e_n)$ be an abbreviation for $s(e_1,\ s(e_2,\ s(\ldots\ e_n)))$. We then construct the following term.

$$\lambda x_1 \ldots x_k.\, \mathtt{LET}\ y = progn(u_1 = v_1,\ \ldots,\ u_n = v_n,\ w)\ \mathtt{IN}\ y$$

We can force the Hindley-Milner procedure to follow this find operation by another sequence of union operations and another find operation by constructing an term of the following form.

$$\lambda x_1 \ldots x_k.\, \mathtt{LET}\ y = progn(\ldots\ w_1)\ \mathtt{IN}\ \mathtt{LET}\ z = progn(y,\ \ldots, w_2)\ \mathtt{IN}\ z$$

This construction can be extended to any sequence of union-find operations.

Although the Hindley-Milner procedure can be forced to simulate union-find, the union-find operations seem not be playing any significant role in the type inference process. It seems possible that by altering step 5 of the procedure to not require the full computation of $\sigma(\tau)$ one might obtain a truly linear time procedure for typability by types of bounded size.

10 Conclusions

The five results in this paper primarily concern recursive types and flow analysis. Three superficially different type systems have been shown to coincide — type inference from flow analysis, type inference based on recursive types, and type inference based on semantic types. These results do not imply there is a single notion of typability for recursive types. In fact there are a variety of notions of typability corresponding to the variety of possible operational semantics for the term language. Typability should guarantee operational safety but the notion of operational safety is sensitive to the choice of semantics. Let `WRONG` be the application $(c(a)\ a)$ where c is a data constructor and let Ω be the divergent term $((\lambda h.\ (h\ h))\ (\lambda h.\ (h\ h)))$. The Heintze analysis [12] accepts the term (Ω `WRONG`) — in the operational semantics operators are evaluated first and hence this term does not generate a run time error. The notion of typability given here, which is the same as that of Aiken-Wimmers-Lakshman for shallow case statements [2], does not exploit order of argument evaluation — it can be viewed as doing an analysis for an operational semantics that evaluates arguments in parallel. However, this system is based on a strict semantics and accepts the term $(c(\Omega)\ a)$ even though this term generates a run time error under lazy interpretation. It is also possible to construct an analysis appropriate for a lazy semantics which rejects $(c(\Omega)\ a)$ but accepts c(`WRONG`). Alternatively one can define a notion of typability such that typability implies safety under arbitrary reductions, i.e., independent of evaluation strategy. This can be done simply by dropping the antecedents of the form $\exists\ u$ from the flow rules in figure 2. The main point of this paper is that each such type system can be formulated in one of three ways — as a flow analysis, as a system for inferring recursive types, or as a system of inferring semantic types.

Flow rules rules seem to provide the simplest, most flexible, and most transparent framework for defining the analyses. For example, the assumption of strict semantics is explicit in the use of antecedents of the form $\exists u$ in the flow rules in figure 2. This is in contrast to the absence of any suggestion of of call by value semantics in the constraints given in section 7, or in the type inference rules given in [2]. The constraints (or inference rules) do not uniquely determine the set of typable terms. In constraint based or rule based formalisms typability is sensitive to the choice of semantic domain. The semantic domain determines the subtype relation between type expressions. The flow rules exhibit no such subtlety. The structural similarity between the semantics in figure 1 and flow rules in figure 2 allow the flow rules to "track" modifications in the operational semantics.

On the other hand, the constraint based approach may have important advantages in the efficiency of polymorphic analysis. This paper has demonstrated that, in contrast with Hindley-Milner typability, typability in the Aiken-Wimmers-Lakshman system is PSPACE hard even for bounded order and arity. However, this worse case result may not be a good indication of performance in practice. Constraint types allow for the representation and simplification of principal types. This may prove to be essential to the construction of practical polymorphic type inference.

Acknowledgments: I would like to thank Nevin Heintze, Jon Riecke, and Alex Aiken for useful discussions concerning this paper.

References

1. A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, June 1993.
2. A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages*, pages 163–173. Association for Computing Machinery, 1994.
3. Alexander Aiken, Dexter Kozen, Moshe Vardi, and Edward Wimmers. The complexity of set constraints. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proc. 1993 Conf. Computer Science Logic (CSL'93)*, volume 832 of *Lect. Notes in Comput. Sci.*, pages 1–17. Eur. Assoc. Comput. Sci. Logic, Springer, September 1993.
4. Alexander Aiken, Dexter Kozen, and Edward Wimmers. Decidability of systems of set constraints with negative constraints. *Infor. and Comput.*, 1994. To appear. Also Cornell University Tech. Report 93-1362, June, 1993.
5. R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. Also in Proc. POPL91.

6. Association for Computing Machinery. *ACM Symposium on Principles of Programming Languages*, 1990.
7. Leo Bachmair, Herald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.
8. W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Proc. 9th Symp. Logic in Computer Science*, pages 128–136. IEEE, July 1994.
9. W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In *Proc. 35th Symp. Foundations of Computer Science*, pages 642–653. IEEE, November 1994.
10. Thom Frühwirth, Ehud Shapiro, Moshe Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 75–83. IEEE Computer Society Press, 1991.
11. M. Gordon. Hol: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis.* Kluwer, 1987.
12. N. Heintze. Set based analysis of ml programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.
13. N. Heintze. Control flow analysis and type systems. In *Second Static Analysis Symposium (SAS95)*, pages 189–206. Springer Verlag, 1995. Lecture Notes in Computer Science 983.
14. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *ACM Symposium on Principles of Programming Languages* [6], pages 197–209.
15. F. Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.
16. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3), March 1995. Also in PLDI-94.
17. S. Jagannathan and A. Write. Effective flow analysis for avoiding run time checks. In *Second Static Analysis Symposium (SAS95)*, pages 207–224. Springer Verlag, 1995. Lecture Notes in Computer Science 983.
18. A. Kfoury, J. Tiuryn, and P. Urzyczyn. Ml typability is dexptime complete. In *Proceedings of the 15th Colloquium on Trees in Algebra and Programming (CAAP)*, pages 468–476. ACM, 1990.
19. D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *POPL84*, pages 165–174, 1984.
20. H. Mairson. Deciding ml typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages* [6].
21. Robin Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.
22. P. Mishra. Towards a theory of types in prolog. In *International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.
23. P. Mishra and U. S. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 7–21. ACM, 1985.
24. Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic.* MIT Press, 1991.

25. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *POPL95*, pages 367–378, 1995.
26. J. Palsberg and S. Smith. Constraned types and their expressivelness. *Transactions on Programming Languages and Systems.* to appear.
27. Olin Shivers. Data flow analysis and type recovery in scheme. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 47–87. MIT Press, 1991.