Static Type Inference in a Dynamically Typed Language

Alexander Aiken
IBM Almaden Research Center
650 Harry Rd.
San Jose, CA 95120
email: *aiken@ibm.com*

Brian R. Murphy
Computer Science Department
Stanford University
Stanford, CA 94305
email: *brm@neon.stanford.edu*

Abstract

We present a type inference system for FL based on an operational, rather than a denotational, formulation of types. The essential elements of the system are a type language based on regular trees and a type inference logic that implements an abstract interpretation of the operational semantics of FL. We use a non-standard approach to type inference because our requirements—using type information in the optimization of functional programs—differ substantially from those of other type systems.

1 Introduction

Compilers derive at least two benefits from static type inference: the ability to detect and report potential run-time errors at compile-time, and the use of type information in program optimization. Traditionally, type systems have emphasized the detection of type errors. *Statically typed* functional languages such as Haskell [HWA*88] and ML [HMT89] include type constraints as part of the language definition, making some type inference necessary to ensure that type constraints are satisfied (i.e., to ensure that a program has no type errors). We, however, are interested in using type inference as a tool for the optimization of programs written in a functional language.

We present a type inference system for FL [B*89], a functional language based on FP [Bac78]. Unlike ML and Haskell, the FL language definition does not include any type constraints. In standard terminology, FL is *dynamically typed*, meaning that run-time type checks are required. However, static type information is still very useful for the compilation of FL programs and particularly for enabling program optimization.

Two characteristics distinguish type inference in our setting from the traditional type inference problem. First, for type information to be useful for optimization, it should be as precise as possible. This is not the case in type inference for ML, for example, where the only requirement is to infer information that is accurate enough to prove that the language's type constraints are satisfied. Second, because FL has no static type restrictions, it is desirable that an FL type inference algorithm infer some useful type information for many programs that other type systems would reject entirely as "untypeable".

Type systems for languages such as ML and Haskell can be characterized roughly as follows. A type is a *set of values*. There is a formal language to describe types and a logic to assign types to the subexpressions of a program. There is usually a particular algorithm, the *type inference* algorithm, for applying the rules of the logic. The correctness of the type inference algorithm is derived from the denotational semantics of the language.

Our type system for FL differs substantially from the standard approach. Nevertheless, there are close parallels, and for this reason we choose to use the terms *type* and *type inference*, despite the non-standard interpretation. For us, a type is a *set of normal-form expressions.* We give a formal language based on *regular trees* [GS84] to describe types. We give a logic to assign types to the subexpressions of a program. Our type inference algorithm applies this logic to carry out an *abstract interpretation* [CC79, Nie85] of the operational semantics of FL. The correctness of this type inference algorithm is established using the operational semantics.

Our approach can be viewed as an operational (or *intensional*) formulation of types. For first-order values (e.g., the atoms, sequences of atoms, etc.) our formulation is equivalent to the standard one, because there is a one-to-one correspondence between the first-order expressions and the values they denote. For higher-order values (e.g., functions and structures containing functions) this correspondence breaks down, and two functions that denote the same value may be members of different (and incomparable) types. As discussed in Section 3, the main reason for adopting this approach is that it appears to provide greater precision on a wider range of FL programs than the standard approach.

Using this operational formulation of types, we construct a type language using regular trees in which type containment is decidable (Section 4). Furthermore, when types are sets of expressions, all the major features of FL, including higher-order functions, sequences, recursive structures, and user-defined datatypes, are precisely captured by regular trees.

Section 5 solves a simplified type inference problem using an abstract interpretation of an operational (rewrite) semantics of FL. The abstract interpretation is presented as a logical inference system. For many recursive functions, proofs in this logic do not exist—i.e., the abstract interpretation does not terminate. Section 6 adds new inference rules to the logic to guarantee termination and to type recursive functions precisely. Section 7 presents a solution to the full type inference problem, using a simple way of defining a *collecting interpretation* [Nie85, HY88] directly from the structure of a proof in the logic. Section 8 concludes with a brief discussion of the implementation and outstanding problems.

This paper concentrates on the theoretical basis for the type system; detailed descriptions of some of the algorithms and the implementation appear in [Mur90]. An implementation of our type inference system for the full FL language has been in use at IBM Almaden for several months. During this time, the system has been used to analyze a wide variety of small- to medium-size (500 lines of FL) programs. Our initial experience indicates that the system is very precise. More importantly, the type information has proven to be of great benefit in program optimization. We believe these results should be applicable to other functional languages, especially those without static type restrictions such as the functional subsets of Lisp and Scheme.

2 An Overview of FL

The FL domain consists of *normal* and *exceptional* values. The normal values are the atoms (integers, reals, characters, and the values *true* and *false*), sequences, tagged normal values, and functions. The exceptional values are specially tagged normal values and $\perp$. The set of all exceptional values is $\mathcal{E}_{\mathrm{FL}}$. Figure 1 gives the subset of FL needed to understand the examples that follow. Some features of the language are ignored altogether; in particular, input/output functions and syntactic sugar are omitted. Note that all FL functions are strict with respect to

$\mathtt{f}$	denotes a function
$\mathtt{f}{:}\mathtt{x}$	denotes function application
$\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle$	denotes sequence construction
$\prec \mathit{integer}, \mathtt{x} \succ$	denotes tag pairs
$\prec \mathit{err}, \mathtt{x} \succ$	denotes exceptions
"abc"	abbreviates the sequence of chars $\langle \text{`}\mathtt{a}, \text{`}\mathtt{b}, \text{`}\mathtt{c} \rangle$

$$\begin{aligned}
\mathtt{comp}{:}\mathtt{x} &= \prec \mathit{err}, \langle \text{"comp"}, \mathtt{x} \rangle \succ \text{ if } \mathtt{x} \neq \langle \mathtt{f}_1, \ldots, \mathtt{f}_n \rangle \\
\mathtt{cons}{:}\mathtt{x} &= \prec \mathit{err}, \langle \text{"cons"}, \mathtt{x} \rangle \succ \text{ if } \mathtt{x} \neq \langle \mathtt{f}_1, \ldots, \mathtt{f}_n \rangle \\
\mathtt{cond}{:}\mathtt{x} &= \prec \mathit{err}, \langle \text{"cond"}, \mathtt{x} \rangle \succ \text{ if } \mathtt{x} \neq \langle \mathtt{f}_1, \mathtt{f}_2, \mathtt{f}_3 \rangle
\end{aligned}$$

$$\begin{aligned}
\mathtt{comp}{:}\langle \mathtt{f}_1, \ldots, \mathtt{f}_n \rangle{:}\mathtt{x} &= \mathtt{f}_1{:}(\ldots(\mathtt{f}_n{:}\mathtt{x})) \\
\mathtt{cons}{:}\langle \mathtt{f}_1, \ldots, \mathtt{f}_n \rangle{:}\mathtt{x} &= \langle \mathtt{f}_1{:}\mathtt{x}, \ldots, \mathtt{f}_n{:}\mathtt{x} \rangle \\
\mathtt{cond}{:}\langle \mathtt{f}_1, \mathtt{f}_2, \mathtt{f}_3 \rangle{:}\mathtt{x} &= \begin{cases} \mathtt{f}_1{:}\mathtt{x} & \text{if } \mathtt{f}_1{:}\mathtt{x} \in \mathcal{E}_{\mathrm{FL}} \\ \mathtt{f}_3{:}\mathtt{x} & \text{if } \mathtt{f}_1{:}\mathtt{x} = \mathit{false} \\ \mathtt{f}_2{:}\mathtt{x} & \text{otherwise} \end{cases} \\
\mathtt{seqof}{:}\mathtt{f}{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle &= \begin{cases} \mathtt{f}{:}\mathtt{x}_\mathtt{i} & \text{min } \mathtt{i} \text{ } \mathtt{s.t.} \text{ } \mathtt{f}{:}\mathtt{x}_\mathtt{i} \in \mathcal{E}_{\mathrm{FL}} \\ \mathit{false} & \exists \mathtt{i} \text{ } \mathtt{s.t.} \text{ } \mathtt{f}{:}\mathtt{x}_\mathtt{i} = \mathit{false} \\ \mathit{true} & \text{otherwise} \end{cases} \\
\mathtt{f}{:}\mathtt{x} &= \mathtt{x} \text{ if } \mathtt{x} \in \mathcal{E}_{\mathrm{FL}} \\
\mathtt{id}{:}\mathtt{x} &= \mathtt{x} \\
\mathtt{K}{:}\mathtt{x}{:}\mathtt{y} &= \mathtt{x} \\
\mathtt{isint}{:}\mathtt{i} &= \mathit{true} \text{ if } i \in \mathit{Int}, \mathit{false} \text{ o.w.} \\
\mathtt{signal}{:}\mathtt{x} &= \prec \mathit{err}, \mathtt{x} \succ \\
*{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle &= \mathtt{x}_1 * \ldots * \mathtt{x}_n \text{ if } \mathtt{x}_\mathtt{i} \in \mathit{Num} \\
\mathtt{si}{:}\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle &= \mathtt{x}_\mathtt{i} \\
\mathtt{tl}{:}\langle \mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \rangle &= \langle \mathtt{x}_2, \ldots, \mathtt{x}_n \rangle \\
\mathtt{al}{:}\langle \mathtt{x}, \langle \mathtt{y}_1, \ldots, \mathtt{y}_n \rangle \rangle &= \langle \mathtt{x}, \mathtt{y}_1, \ldots, \mathtt{y}_n \rangle
\end{aligned}$$

Figure 1: A subset of FL.

exceptions. We use the following abbreviations:

$$\begin{aligned} \mathtt{f} \circ \mathtt{g} &= \mathtt{comp}\colon \langle \mathtt{f}, \mathtt{g} \rangle \\ [\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{n}] &= \mathtt{cons}\colon \langle \mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{n} \rangle \\ \mathtt{p} \rightarrow \mathtt{q}; \mathtt{r} &= \mathtt{cond}\colon \langle \mathtt{p}, \mathtt{q}, \mathtt{r} \rangle \end{aligned}$$

Some functions in Figure 1 are defined only for some arguments; for all other arguments, the application $\mathtt{f}:\mathtt{x}$ returns the exception $\prec err, <$ "f"$, \mathtt{x} >\succ$. The evaluation order of FL is leftmost-innermost; thus, in $[\mathtt{f}, \mathtt{g}]:\mathtt{x}$, $\mathtt{f}:\mathtt{x}$ is evaluated and then $\mathtt{g}:\mathtt{x}$ is evaluated.

The function definitions in Figure 1 can be interpreted as *rewrite rules*, where the rewriting consists of substituting the right-hand side of the definition for the left-hand side. An expression is in *normal form* if no rewrite rule applies. Examples of normal form expressions are: $\mathtt{1}$, $\mathtt{id}$, $\langle \mathtt{1}, \mathtt{id} \rangle$, $\mathtt{K} \circ \mathtt{id}$, and $\prec err, <$ "f"$, \mathtt{x} >\succ$. Note that exceptions are legitimate normal forms.

3 The Type Inference Problem

The following example shows a very simple use of type information in program optimization. Consider an expression $E(*)$. If, in the context of $E()$, the function $*$ is guaranteed to be applied to sequences of integers, then $*$ can be replaced by integer multiplication $*_{\mathrm{int}}$, which is a faster operation.

This small example illustrates certain characteristics that allow a type system to be useful in optimization: the type system should associate with each function f in a program a set of possible arguments, f_{arg}. (It also useful to have a set of possible results; the necessary modifications to the type system are simple and omitted.) To be correct, type information must be *conservative*—f_{arg} must be a superset of the actual set of possible arguments of f.[1] Finally, it should be possible to compare types, and in particular to test whether one type is a subset of another type. In the example above, the precondition for the transformation is that $f_{arg} \subseteq$ *sequences_of_integers*.

3.1 Types as Sets of Values

Let e be an expression, let μ :: Expressions $\rightarrow$ Values be the meaning function mapping expressions to values, and let a type T be a set of values. The statement "e has type T" is formalized as the assertion $\mu(e) \in T$.

In type systems where types are sets of values, typing an application $e_1\colon e_2$ involves computing a type T_1 for e_1 and T_2 for e_2, and then applying a rule $App(T_1, T_2)$ to derive a type for $e_1\colon e_2$. In general, the rule App may constrain the types T_1 and T_2, thereby gaining information about the behavior of e_1 and e_2 in the context $e_1\colon e_2$.

For example, in $\mathtt{cond}\colon \langle \mathtt{seqof}\colon \mathtt{isint}, *, \mathtt{signal} \rangle$ the argument of $*$ is always a sequence of integers; thus $*_{\mathtt{int}}$ can replace $*$, as discussed above, provided the type information for $*$ is precise. Suppose $\mathtt{cond}$ has type T_1 and $\langle \mathtt{seqof}\colon \mathtt{isint}, *, \mathtt{signal} \rangle$ has type T_2. For $App(T_1, T_2)$ to tightly constrain the type of $*$'s argument, the type T_1 should express the following properties: $\mathtt{cond}$ takes a sequence of functions $\langle \mathtt{f}_1, \mathtt{f}_2, \mathtt{f}_3 \rangle$ as its first argument; $\mathtt{f}_1$ is applied to $\mathtt{cond}$'s second argument to determine which arm of the conditional to use; and finally, $\mathtt{f}_2$ is applied

[1]A formal definition of conservative requires more development; see Section 7.

TE	::=	$\emptyset$	$\Psi(\emptyset, \sigma)$	$=$	$\{\}$
	$\mid$	α	$\Psi(\alpha, \sigma)$	$=$	$\sigma(\alpha)$
	$\mid$	$\mathrm{TE}_1 \vee \mathrm{TE}_2$	$\Psi(\mathrm{TE}_1 \vee \mathrm{TE}_2, \sigma)$	$=$	$\Psi(\mathrm{TE}_1, \sigma) \cup \Psi(\mathrm{TE}_2, \sigma)$
	$\mid$	$\mathrm{TE}_1 \wedge \mathrm{TE}_2$	$\Psi(\mathrm{TE}_1 \wedge TE_2, \sigma)$	$=$	$\Psi(\mathrm{TE}_1, \sigma) \cap \Psi(\mathrm{TE}_2, \sigma)$
	$\mid$	$\mathit{fix}\ \alpha.\mathrm{TE}_1$	$\Psi(\mathit{fix}\ \alpha.\mathrm{TE}_1, \sigma)$	$=$	least T s.t. $T = \Psi(\mathrm{TE}_1, \sigma[\alpha \leftarrow T])$
	$\mid$	$c(\mathrm{TE}_1, \ldots, \mathrm{TE}_{\mathrm{arity}(c)})$	see Figure 3		

Figure 2: Syntax and semantics of type expressions.

to values for which `f1` is true. Thus, if we wish to infer precise information for $*$, the type of `cond` is highly constrained—ideally, the type should be $\{\mu(\texttt{cond})\}$ to avoid losing information.

Similar arguments can be constructed for many of the primitive functions using simple and typical programs. Within the framework of types as sets of values, the most general solution to this problem is to admit arbitrary functions from types to types as types; this allows unlimited precision in the types of primitive functions. However, if types are arbitrary functions, it is very difficult to test whether $T_1 \subseteq T_2$. As we shall see, the ability to compare types is important not only for enabling optimization, but also for the precision of type inference itself.

Our solution to this dilemma is to redefine types as *sets of expressions*. The type of `cond` becomes just the set of expressions {`cond`}, and knowledge about the behavior of `cond` is captured through a separate mechanism of *type rewrite rules*. The advantage of this approach is that types have simple semantics (making it possible to test whether $T_1 \subseteq T_2$) while still allowing arbitrary precision (embodied in the type rewrite rules).

3.2 Related Work

It is interesting to examine the effect defining types as sets of values has had on other type systems for dynamically typed languages. Thatte presents a type system for Lisp designed to detect statically some type errors [Tha88]. The system extends the Hindley/Milner type system [Mil78] with one new type Ω representing the set of all values; thus, every expression has type Ω. This is probably the simplest extension of Hindley/Milner that is adequate for dynamically typed languages. Unfortunately, the typing algorithm Thatte presents for this minimal extension may diverge. Furthermore, the system cannot precisely type some simple expressions; for example, the expression `s1`:$\langle$`1`, `a`$\rangle$ has type Ω because heterogeneous sequences have type *List*(Ω).

At the other extreme, Young and O'Keefe present a *type evaluator* for a lazy dialect of Scheme [YO88]. This system is strikingly similar to our system in some respects, but types are sets of values. Scheme itself serves as the type language; thus, any function from types to types is a type. As discussed above, this presents serious problems in comparing types. Young and O'Keefe use a conservative test for type equality. In contrast, a different definition of type leads us to a more robust subset test. Other algorithms on types (such as type intersection and union) are also correspondingly weaker in their approach because of the difficulty of handling function types.

Several type systems have been proposed for FP [GHW81, Fra81, Kat84]. The drawbacks of using these approaches for FL type inference are the essential first-order nature of the systems and the inability to automatically infer types for recursive data structures.

Regular trees have been used before in a type language for a statically-typed, first-order functional language [MR85]. There are two essential differences in our approach. First, in our system regular trees are the only

TE	::=	$\mathrm{TE}_1 \mapsto \mathrm{TE}_2$	$\Psi(\mathrm{TE}_1 \mapsto \mathrm{TE}_2, \sigma) = \{\langle e_0, \ldots, e_n\rangle \mid e_0 \in \Psi(\mathrm{TE}_1, \sigma) \wedge \langle e_1, \ldots, e_n\rangle \in \Psi(\mathrm{TE}_2, \sigma)\}$
	|	$\mathrm{TE}_1{:}\mathrm{TE}_2$	$\Psi(\mathrm{TE}_1{:}\mathrm{TE}_2, \sigma) = \{e_1{:}e_2 \mid e_1 \in \Psi(\mathrm{TE}_1, \sigma) \wedge e_2 \in \Psi(\mathrm{TE}_2, \sigma)\}$
	|	$\prec tag, \mathrm{TE}_1 \succ$	$\Psi(\prec tag, \mathrm{TE}_1 \succ, \sigma) = \{\prec tag, e \succ \mid e \in \Psi(\mathrm{TE}_1, \sigma)\}$
	|	Null	$\Psi(\mathrm{Null}, \sigma) = \{\langle\rangle\}$
	|	Ω	$\Psi(\Omega, \sigma) = \{e \mid e \text{ is a normal form}\}$
	|	Int	$\Psi(\mathrm{Int}, \sigma) = \{\ldots, -1, 0, 1, \ldots\}$
	|	$\overline{\mathrm{cond}}$	$\Psi(\overline{\mathrm{cond}}, \sigma) = \{\mathrm{cond}\}$

Figure 3: The constructors.

mechanism for expressing types; in [MR85] a separate function-space constructor is used for function types. Second, we require arbitrary type union, whereas [MR85] uses discriminated unions. Regular trees have also been used in the static analysis of logic programs [HJ90].

4 The Type Language

Let Σ be a set of type constructors and let Δ be a set of type variables. We use c for elements of Σ and $\alpha, \beta, \ldots$ for elements of Δ. A syntax and semantics for type expressions is given in Figure 2. Let T be a type expression. An occurrence of α in T is *bound* if it is in the scope of *fix* $\alpha.T'$; otherwise α is *free*. Except for the presence of free variables, type expressions are *regular trees* [GS84]. An *environment* σ is a function from type variables to sets of expressions. The type meaning function Ψ maps type expressions and environments to sets of expressions.

Definition 4.1 Let T_1 and T_2 be type expressions. Then $T_1 \subseteq T_2$ if $\forall\sigma\ \Psi(T_1, \sigma) \subseteq \Psi(T_2, \sigma)$, and $T_1 = T_2$ if $T_1 \subseteq T_2$ and $T_2 \subseteq T_1$.

Recall from Section 3 that the goal of type inference is to assign conservative argument types to every function in a program. Since FL has call-by-value semantics, the argument of a function is always in normal form. This leads to the following definition of type.

Definition 4.2 Let Ω be the set of all normal forms, let τ be the substitution such that $\forall\alpha\ \tau(\alpha) = \Omega$, and let T be a type expression. T is a type if $\Psi(T, \tau) \subseteq \Omega$.

From this point, we use e for arbitrary expressions, v for expressions in normal form, T for type expressions, and V for types. We omit environments and write $e \in T$ for $e \in \Psi(T, \sigma)$ when a statement is quantified over all environments σ.

A subset of the constructors for our type language is given in Figure 3. We use infix notation for constructors to emphasize the relationship with the syntax of FL (Figure 1). In addition to the constructors listed, there is a constructor $\overline{f}$ for each primitive function f. There are also constructors covering the atoms: Char, Int, Real, and False are used in examples. Other useful types are listed in Figure 4.

Type expressions precisely describe common programming structures. Every primitive function f is exactly described, since $\overline{f} = \{f\}$. Applications are also captured; for example, the expression $\texttt{cond}{:}\langle\texttt{seqof}{:}\texttt{isint}, *, \texttt{id}\rangle$ is the only element of the type $\overline{\texttt{cond}}{:}[\![\overline{\texttt{seqof}}{:}\overline{\texttt{isint}}, \overline{*}, \overline{\texttt{id}}]\!]$. Regular trees naturally describe recursive structures.

$$\begin{array}{lcl@{\quad}rcl}
\text{Num} & = & \text{Int} \vee \text{Real} & \text{Exc} & = & \prec err, \Omega \succ \\
\text{True} & = & \neg(\text{False} \vee \text{Exc}) & \text{Seq} & = & \textit{fix}\ \alpha.\text{Null} \vee (\Omega \mapsto \alpha) \\
\text{Func} & = & \{v \mid v \text{ is a function }\} & [\![T_1, \ldots, T_n]\!] & = & T_1 \mapsto (\ldots \mapsto (T_n \mapsto \text{Null}) \ldots)
\end{array}$$

Figure 4: Some useful type expressions.

$$\frac{}{*\!: v \ \rightarrow_{\text{FL}} \begin{cases} 1 * v_1 * \ldots * v_n & \text{if } v = \langle v_1, \ldots, v_n \rangle \ \ v_i \in \text{Num} \\ \prec err, < \text{“}*\text{”}, v >\succ & \text{otherwise} \end{cases}} \qquad \text{MULT}$$

$$\underset{\text{CONDERR}}{\frac{v \neq \langle v_1, v_2, v_3 \rangle \vee \exists i \ v_i \notin \text{Func}}{\mathtt{cond}\!: v \ \rightarrow_{\text{FL}} \ \prec err, \text{“cond”}, v >\succ}} \qquad \underset{\text{COND1}}{\frac{v_1\!: v_4 \ \rightarrow_{\text{FL}} \ v_5, \quad v_5 \in \text{Exc}}{\mathtt{cond}\!: \langle v_1, v_2, v_3 \rangle\!: v_4 \ \rightarrow_{\text{FL}} \ v_5}}$$

$$\underset{\text{COND2}}{\frac{v_1\!: v_4 \ \rightarrow_{\text{FL}} \ v_5, \quad v_5 \in \text{True}, \quad v_2\!: v_4 \ \rightarrow_{\text{FL}} \ v_6}{\mathtt{cond}\!: \langle v_1, v_2, v_3 \rangle\!: v_4 \ \rightarrow_{\text{FL}} \ v_6}} \qquad \underset{\text{COND3}}{\frac{v_1\!: v_4 \ \rightarrow_{\text{FL}} \ v_5, \quad v_5 \in \text{False}, \quad v_3\!: v_4 \ \rightarrow_{\text{FL}} \ v_6}{\mathtt{cond}\!: \langle v_1, v_2, v_3 \rangle\!: v_4 \ \rightarrow_{\text{FL}} \ v_6}}$$

Figure 5: Sample FL rewrite rules.

The set of sequences of type V (henceforth Seqof(V)) is $\textit{fix}\ \alpha.\text{null} \vee V \mapsto \alpha$. Binary trees with leaves of type V are $\textit{fix}\ \alpha.[\![\alpha, \alpha]\!] \vee V$. More interestingly, a type expression for an iterated constant function that (eventually) returns an element of V is $\textit{fix}\ \alpha.\overline{\mathtt{K}}\!: (\alpha \vee V)$. Finally, type expressions support unrestricted type union. This is necessary for the precise typing of conditionals, for example, since the types of the two branches need not be the same.

Type expressions have many useful properties. From Figure 2 type expressions are closed under union, intersection, and least fixed-point operations. Type expressions with no free variables are closed under complementation. It is also decidable whether $T = \emptyset$ [GS84] and whether $T_1 \subseteq T_2$ [Mur90]. We have developed a fast heuristic (based on the work of [MR85]) for the following problem: Given two types T_1 and T_2, find a most general substitution σ such that $\Psi(T_1, \sigma) \subseteq \Psi(T_2, \sigma)$. A heuristic is required because finding a substitution is PSPACE-hard in general [Mur90]. The heuristic works well; we have yet to find an example in practice where its use adversely affects the precision of type inference.

We conclude this section with two brief comments on the implementation of types. First, for increased precision, our system propagates an atom (e.g., 1) instead of its type (e.g., Int) insofar as possible.[2] For clarity, atoms are used in types in the examples. Second, while the definition of type expressions given in Figure 2 is easy to understand, it does not lead to the best algorithms. An alternative representation, known as *regular* ΣX*-grammars* [GS84] or *leaf-linear systems of equations* [MR85], is used in our implementation.

5 Type Rewriting

Our type inference algorithm is an *abstract interpretation* [CC79, Nie85] of an operational semantics of FL. The correctness of type inference is established by comparing two rewrite relations: the standard evaluation relation

[2] Including atoms as types prevents a crisp statement of some results; thus, the exclusion is for pedagogical reasons.

$$\frac{\forall i \le n\ e_i \rightarrow_{\text{FL}} v_i}{e \rightarrow_{\text{FL}} v} \text{ where} \qquad\qquad \frac{\forall i \le n\ T_i \rightarrow_{\text{T}} V_i}{T \rightarrow_{\text{T}} V} \text{ where}$$

$$\begin{array}{rcl} e_i & = & E_{i-1}(e, v_1, \ldots, v_{i-1}) \\ v & = & E_n(e, v_1, \ldots, v_n) \end{array} \qquad\qquad \begin{array}{rcl} T_i & = & \mathcal{E}_{i-1}(T, V_1, \ldots, V_{i-1}) \\ V & = & \mathcal{E}_n(T, V_1, \ldots, V_n) \end{array}$$

Figure 6: The general form of an FL rewrite rule and its type rewrite rule abstraction.

$\rightarrow_{\text{FL}}$ on FL expressions, and an abstract evaluation relation $\rightarrow_{\text{T}}$ on type expressions. This section solves a simplified type inference problem: Define $\rightarrow_{\text{T}}$ on type expressions such that when $e \in T$, $e \rightarrow_{\text{FL}} v$, and $T \rightarrow_{\text{T}} V$, then $v \in V$. Section 6 alters the abstract rewrite relation to guarantee termination and to precisely type recursive functions. Section 7 presents a full solution to the type inference problem based on the abstract rewrite relation.

We begin by writing the operational rewrite rules of FL in the structural style advocated by Plotkin [Plo]. Rewrite rules in this style are presented as inference rules of a formal logic; a rule is read as asserting that if the subgoals above the line hold, then the conclusion below the line holds. A proof of $e \rightarrow_{\text{FL}} v$ in this logic is a computation tree showing the evaluation of e. Examples of FL rewrite rules for the functions $*$ and `cond` are given in Figure 5.

Type rewrite rules rewrite type expressions to types. Type rewrite rules are given in the same style as FL rewrite rules, with one type rewrite rule corresponding to each FL rewrite rule. The general form of an FL rewrite rule and its corresponding type rewrite rule is given in Figure 6. (The constraint that the number of subgoals be the same in corresponding FL and type rewrite rules can always be satisfied by padding rules with "dummy" subgoals.) The following definition provides a condition for the correctness of a type rewrite rule.

Constraint 5.1 (Conservative Rules) Let R_{FL} and R_{T} be corresponding FL and type rewrite rules as shown in Figure 6. R_{T} is *conservative* with respect to R_{FL} if

$$\begin{array}{ll} \forall i, T, V_1, \ldots, V_{i-1} & \{E_i(e, v_1, \ldots, v_{i-1}) | e \in T, v_j \in V_j\} \\ & \subseteq \mathcal{E}_i(T, V_1, \ldots, V_{i-1}) \end{array}$$

If all type rewrite rules satisfy this constraint, then type rewriting is conservative with respect to the standard semantics.

Theorem 5.2 Suppose $e \rightarrow_{\text{FL}} v$ and $T \rightarrow_{\text{T}} V$. If type rewrite rules are conservative and $e \in T$, then $v \in V$.

Sample type rewrite rules for $*$ and `cond` are given in Figure 7. In practice, we have found that precise type inference requires sophisticated type rewrite rules such as TCOND. For precision, it is important to know for which arguments the functions in V_1 are true and false. To approximate this, we have implemented a conservative *type inversion* function with the property

$$V^{-1}(V') \supseteq \{v | v'' : v \rightarrow_{\text{FL}} v', v' \in V', v'' \in V\}$$

Using TCOND, and others rules not specified here, the type system proves the following:

$$\overline{cond}{:}\,[\![\overline{seqof}{:}\,\overline{isint}, \overline{*}, \overline{signal}]\!]{:}\,\Omega \quad \rightarrow_{\text{T}}$$
$$\text{Int} \vee \prec err, \neg\text{Seqof(Int)} \succ$$

$$\frac{}{\overline{*}{:}V \ \rightarrow_{\mathrm{T}} \ \begin{array}{r} (\text{IF } V \wedge \text{Seqof(Int)} \neq \emptyset \text{ THEN Int ELSE } \emptyset) \quad \vee \\ (\text{IF } V \wedge \text{Seqof(Num)} \wedge \neg\text{Seqof(Int)} \neq \emptyset \text{ THEN Num ELSE } \emptyset) \quad \vee \\ \prec err, [\![\text{``}*\text{''}, V \wedge \neg\text{Seqof(Num)}]\!] \succ \end{array}} \quad \text{TMULT}$$

$$\frac{V \not\subseteq X \text{ where} X = [\![\text{Func}, \text{Func}, \text{Func}]\!]}{\overline{\texttt{cond}}{:}V \ \rightarrow_{\mathrm{T}} \ \overline{\texttt{cond}}{:}(V \wedge X) \vee \prec err, < \text{``cond''}, V \wedge \neg X > \succ} \quad \text{TCONDERR}$$

$$\frac{V_1{:}V_4 \ \rightarrow_{\mathrm{T}} \ V_5, \ V_2{:}(V_1^{-1}(\text{True}) \wedge V_4) \ \rightarrow_{\mathrm{T}} \ V_6, \ V_3{:}(V_1^{-1}(\text{False}) \wedge V_4) \ \rightarrow_{\mathrm{T}} \ V_7}{\overline{\texttt{cond}}{:}[\![V_1, V_2, V_3]\!]{:}V_4 \ \rightarrow_{\mathrm{T}} \ (V_5 \wedge \text{Exc}) \vee V_6 \vee V_7} \quad \text{TCOND}$$

Figure 7: Sample type rewrite rules.

$$\frac{}{T \rightarrow_{\mathrm{T}} \Omega} \qquad \frac{\forall i \leq n \ \ A \cup \{T' \rightarrow_{\mathrm{T}} V'\} \vdash T_i \rightarrow_{\mathrm{T}} V_i \ \ T' \subseteq T \ \ V \subseteq V'}{A \vdash T \rightarrow_{\mathrm{T}} V} \qquad \frac{}{A \cup \{T \rightarrow_{\mathrm{T}} V\} \vdash T \rightarrow_{\mathrm{T}} V}$$

TERM REC ASSUME

Figure 8: Type rewrite rules to guarantee termination.

6 Termination

An implementation of type rewriting as defined so far would be correct but would also frequently fail to terminate. Consider the following function, which is the identity on sequences:

$$\texttt{def f} \equiv \texttt{isnull} \rightarrow [\,]; \texttt{al} \circ [\texttt{s1}, \texttt{f} \circ \texttt{tl}]$$

This example introduces FL's function definition mechanism. We assume, without loss of generality, that the right hand side of a function definition is a function in normal form. For each function definition $\texttt{def f} \equiv \texttt{v}$ in a program, a constructor $\overline{f}$ is added to the type language, a type V is computed such that $v \in V$, and a new type rewrite rule $\overline{f} \rightarrow_{\mathrm{T}} V$ is defined.

Consider the type application $\overline{\texttt{f}}{:}\text{Seq}$. A proof P of $\overline{\texttt{f}}{:}\text{Seq} \rightarrow_{\mathrm{T}} V$ must correspond, step for step, with proofs p_v of $\texttt{f}{:}v \rightarrow_{\mathrm{FL}} v'$ for every $v \in \text{Seq}$. But there is no finite proof P, since there is no bound on the size of all p_v. In an implementation, type rewriting diverges searching for the proof P.

Figure 8 defines new type rewrite rules TERM and ASSUME; REC is a scheme for modifying all type rewrite rules of the form given in Figure 6. The rule TERM can always be applied to prevent type rewriting from diverging. However, using TERM usually results in a great loss of information, so it is used only if other techniques fail.

The most important technique for guaranteeing the termination of type rewriting is embodied in REC and ASSUME. In general, to use a REC rule a substitution σ is needed such that $T'\sigma \subseteq T\sigma$ and $V\sigma \subseteq V'\sigma$. The system tries to find a *most general* substitution σ that maximizes the domain T of the type rewriting and minimizes the range V.

Our implementation uses several heuristics to discover a valid assumption $T' \rightarrow_{\mathrm{T}} V'$ for REC. We illustrate the most important heuristic using the function $\texttt{f}$ above. In evaluating $\overline{\texttt{f}}{:}\text{Seq}$, the system first discovers that this

$$\frac{S \supseteq \{V\} \cup \{X | \exists Y \in S \text{ s.t. } \mathit{Fun}(Y){:}\,\Omega_L \rightharpoonup_{\mathrm{T}} X\}}{\Omega_L{:}\,V \rightharpoonup_{\mathrm{T}} \Omega_L \vee \bigvee_{X \in S} X} \qquad \text{CONTEXT}$$

where $\mathit{Fun}(V) = \{f | f \text{ is a subexpression of } e \in V\}$

$$\frac{}{T \rightharpoonup_{\mathrm{T}} \Omega_{Lab(T)}} \qquad \text{TERM}$$

Figure 9: Additional type rewrite rules.

$$H :: \text{Label} \times \text{Int} \rightarrow \text{Type}$$

$$\begin{aligned} (H_1 \sqcup H_2)(l,n) &= H_1(l,n) \cup H_2(l,n) \\ H_1 \leq H_2 &\Leftrightarrow \forall l,n\ H_1(l,n) \subseteq H_2(l,n) \end{aligned}$$

$$H_e(l,n) = \{v_n | v_n \neq \bot \wedge l.f{:}\,v_1{:}\ldots{:}\,v_n \in \bigcup_{P \in \mathcal{P}_e} \mathit{Conc}(P)\}$$

$$H_T(l,n) = \begin{cases} \Omega_L & \text{if } T' \rightharpoonup_{\mathrm{T}} \Omega_L \in \mathit{Conc}(P_T) \wedge l \in L \text{ where } L = \mathit{Lab}(T') \\ \bigcup\{V_n | l.f{:}\,V_1{:}\ldots{:}\,V_n \in \mathit{Conc}(P_T)\} & \text{otherwise} \end{cases}$$

Figure 10: History functions.

recursively involves the computation of $\overline{\texttt{f}}$:Seq, when the occurrence of `f` in the body is applied. Since $\overline{\texttt{f}}{:}\text{Seq} \subseteq \overline{\texttt{f}}{:}\text{Seq}$ the system adds the assumption $\overline{\texttt{f}}{:}\,\text{Seq} \rightharpoonup_{\mathrm{T}} \alpha$, where α is a new type variable, and applies ASSUME. In the last step of the proof, the system has derived

$$\begin{aligned} &\{\overline{\texttt{f}} : \text{Seq} \rightharpoonup_{\mathrm{T}} V'\} \vdash \overline{\texttt{f}} : \text{Seq} \rightharpoonup_{\mathrm{T}} V \quad \text{where} \\ &V = \alpha \\ &V' = \text{Null} \vee (\Omega \mapsto (\alpha \wedge \text{Seq})) \vee \prec \mathit{err}, \langle\text{“al”}, \alpha \wedge \neg\text{Seq}\rangle \succ \end{aligned}$$

At this point, to apply a REC rule a substitution σ is needed such that $V\sigma \subseteq V'\sigma$. In this case the system finds the most general substitution $[\alpha \leftarrow \mathit{fix}\ \alpha.V]$. Since $\mathit{fix}\ \alpha.V = \text{Seq}$, the final result is a proof that $\overline{\texttt{f}}{:}\text{Seq} \rightharpoonup_{\mathrm{T}} \text{Seq}$.

Note that the type V precisely describes the possible outcomes of the body of `f` under the assumption that `f` returns type α. In particular, V shows that the body of `f` produces a modified sequence for every sequence in α, and an append-left (“al”) exception for every non-sequence in α. The application of REC in the final step proves that no append-left exceptions can arise.

7 Type Inference

A *collecting interpretation* [Nie85, HY88] is an abstract interpretation that gathers information about the subexpressions of an expression. This section uses proofs in the logic $\rightharpoonup_{\mathrm{T}}$ and a simple way of defining collecting interpretations to solve the type inference problem stated in Section 3: to determine constraints upon every function's use in a given context. Without loss of generality, we assume that the context is a normal form expression.

We first formalize the result of type inference. Following [HY88], we extend FL's syntax to distinguish occurrences of a primitive function f by labelling each with a unique label l (writing $l.f$).[3] For higher-order functions, it is useful to have type information for every curried argument, not just the first. Thus, the result of type inference is a *history* function (see Figure 10) which associates a type with each label l and argument number n.

It is straightforward to modify the FL and type rewrite rules to propagate, but otherwise ignore, labels. Introducing labels changes the definition of types, however: $l_1.f$ and $l_2.f$ are distinct expressions if $l_1 \neq l_2$, since we wish to track occurrences of functions individually. Define $Lab(e)$ (resp. $Lab(T)$) to be the set of labels in e (resp. T). Let L be a set of labels. We define a family of types $\Omega_L = \{v | Lab(v) \subseteq L\}$.

The next step is to identify the proofs in $\rightarrow_{\mathrm{T}}$ that express the type information we wish to capture about functions in v. Type information must be computed for all surrounding contexts in which v might be used; that is, type information is needed for $v' \!:\! v$ for all contexts v'. The rule CONTEXT in Figure 9 accomplishes this; it analyzes not just v, but also any components of v that are accessible (e.g., if v is a sequence of functions) and any functions that v might return (if v is a higher-order function). In general, finding a finite set S in rule CONTEXT requires use of the rule TERM to ensure closure, although this is rare in practice. For increased accuracy in the presence of labels, the rule TERM is modified as shown in Figure 9.

The conclusion of rule CONTEXT may seem strange for a precise type system; after all, Ω_L contains every computable function. We are not interested in the conclusion, however, but in the structure of the proof itself and what it shows about the possible arguments of functions. The rest of this section develops two history functions, H_e for proofs in $\rightarrow_{\mathrm{FL}}$ and H_T for proofs in $\rightarrow_{\mathrm{T}}$. The following definition formalizes a correctness condition for H_T.

Definition 7.1 (Conservative) Let T be a type expression and let H_T be a history function. H_T is *conservative* if $H_T \geq \bigsqcup_{e \in T} H_e$

We first define a history function for $\rightarrow_{\mathrm{FL}}$. This is complicated by the fact that no proof of the form $e \rightarrow_{\mathrm{FL}} v$ exists in general (i.e., e may diverge). To account for histories of non-terminating computations, a new rewrite rule $e \rightarrow_{\mathrm{FL}} \perp$ is added and all other are rules are modified to enforce strictness with respect to $\perp$. The modifications are easy and omitted for lack of space. Let $\mathcal{P}_e$ be the set of all proofs of $e \rightarrow_{\mathrm{FL}} v$, and let $Conc(P)$ be the set of all conclusions $v_1 \!:\! v_2 \rightarrow_{\mathrm{FL}} v_3$ in a proof P. The history function H_e defined in Figure 10 is precise in that all the steps of the proof are recorded (thus the name "history").

Let P_T be the proof of the form $\vdash T \rightarrow_{\mathrm{T}} V$ discovered by the type system. The definition of a history function H for $\rightarrow_{\mathrm{T}}$ is given in Figure 10. The only difficulty is the rule TERM; the most precise history information inferable from $T \rightarrow_{\mathrm{T}} \Omega_{Lab(T)}$ is that functions in T could be applied to any argument.

Theorem 7.2 The definition of H_T given in Figure 10 is conservative.

Let v be an FL expression in which every primitive function has a unique label. The result of type inference is the history function H_T, where $T = \Omega_{\emptyset} \!:\! V$ and $v \in V$.

[3]Labels may be drawn from any countable set.

8 Conclusions and Future Work

We have presented a type inference algorithm for FL based on an operational view of types where types are sets of expressions. This approach inherently sacrifices some of the potential power in a system where types are sets of values. For example, in our system there is no direct way to prove any relationship between the functions `id` and `id` $\circ$ `id` , even though these expressions denote the same value. However, a system such as ours requires heuristics at some level, since perfectly precise information is uncomputable. Our implementation has convinced us that this design is a good one; this approach seems to match very well with programming styles used in practice.

The main barrier to a completely practical system is performance. The current implementation tries to make the proof search fast. Memoization [Mic68] is used extensively to avoid recomputation and there is no backtracking. However, the underlying algorithm is still exponential in the worst case, since the size of the proof may be exponential in the size of the original program.

A promising solution to this problem is to adapt the idea of *principal types* from statically typed languages [DM82]. Consider the function `f` from Section 6. At present, the system must derive the entire proof `f`: $\mathrm{Seqof}(V) \rightarrow_{\mathrm{T}} \mathrm{Seqof}(V)$ every time it is needed. Instead, the system could prove once that `f`: $\mathrm{Seqof}(\alpha) \rightarrow_{\mathrm{T}} \mathrm{Seqof}(\alpha)$ and then specialize this fact when needed. This introduces no new ideas; the system is already powerful enough to prove `f`: $\mathrm{Seqof}(\alpha) \rightarrow_{\mathrm{T}} \mathrm{Seqof}(\alpha)$. The difficulty is that, in general, there is no "most general" rewriting `f`: $V \rightarrow_{\mathrm{T}} V'$ of `f` for which every other rewriting is obtainable by substitution. The problem, then, is to identify heuristics which accurately predict when the potential cost in precision is small enough that it is worthwhile to use a specialization of a general fact.

9 Acknowledgements

The authors would like to thank the FL group (John Backus, Thom Linden, Peter Lucas, Paul Tucker, John Williams, and Ed Wimmers) for many useful discussions and a congenial working environment. Discussions with Ed Wimmers helped to clarify many of the semantic issues. Paul Tucker assisted with the implementation. It is also a pleasure to thank Carl Gunter, John Mitchell, Uday Reddy, Moshe Vardi, Jennifer Widom, John Williams, and Ed Wimmers for their comments on earlier versions of this paper.

References

[B*89] J. Backus et al. *FL Language Manual, Parts 1 and 2.* Research Report RJ 7100, IBM, 1989.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:8, 1978.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, January 1979.

[DM82] L. Damas and R. Milner. Principle type-schemes for functional programs. In *Proceedings of the Ninth Annual ACM Symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[Fra81] G. Frank. Specification of data structures for FP programs. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 221–228, 1981.

[GHW81] J. Guttag, J. Horning, and J. Williams. FP with data abstraction and strong typing. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 11–24, 1981.

[GS84] F. Gecseg and M. Steinby. *Tree Automata.* Academei Kaido, Budapest, 1984.

[HJ90] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *POPL17*, pages 197–209, January 1990.

[HMT89] R. Harper, R. Milner, and M. Tofte. *The Definition of Standard ML—Version 3.* Technical Report ECFS-LFCS-89-81, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[HWA*88] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, J. Hughes, T. Johnsson, D. Kieburtz, S. P. Jones, R. Nikhil, M. Reeve, D. Wise, and J. Young. *Report on the Functional Programming Language Haskell.* Technical Report DCS/RR-666, Yale University, December 1988.

[HY88] P. Hudak and J. Young. A collecting interpretation of expressions (without powerdomains). In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages*, pages 107–118, 1988.

[Kat84] T. Katayama. Type inference and type checking for functional languages: a reduced computation approach. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 263–272, August 1984.

[Mic68] D. Michie. 'Memo' functions and machine learning. *Nature*, (218):19–22, April 1968.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Comp. & Sys. Sci.*, 17:348–375, 1978.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Proceedings of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 7–21, 1985.

[Mur90] B. R. Murphy. *A Type Inference System for FL.* Master's thesis, MIT, 1990.

[Nie85] F. Nielson. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems*, 7(3):359–379, July 1985.

[Plo] G. D. Plotkin. A structural approach to operational semantics. Text prepared at University of Aarhus.

[Tha88] S. Thatte. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium*, pages 615–629, Springer-Verlag Lecture Notes in Computer Science, vol. 317, July 1988.

[YO88] J. Young and P. O'Keefe. Experience with a type evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 573–581, North-Holland, 1988.