Soft Typing with Conditional Types

Alexander Aiken* and Edward L. Wimmers
IBM Almaden Research Center
650 Harry Rd., San Jose, CA 95120
{aiken,wimmers}@almaden.ibm.com

T. K. Lakshman†
Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave,
Urbana, IL 61801
lakshman@cs.uiuc.edu

Abstract

We present a simple and powerful type inference method for dynamically typed languages where no type information is supplied by the user. Type inference is reduced to the problem of solvability of a system of type inclusion constraints over a type language that includes function types, constructor types, union, intersection, and recursive types, and conditional types. Conditional types enable us to analyze control flow using type inference, thus facilitating computation of accurate types. We demonstrate the power and practicality of the method with examples and performance results from an implementation.

1 Introduction

Most modern programming languages employ type checking to guarantee that functions are applied only to appropriate arguments. Languages differ in the degree to which the type checking is *static* (performed at compile-time) or *dynamic* (performed at run-time). *Statically typed* languages, such as ML, require that function applications be proven type-safe at compile-time. This is enforced by a type inference algorithm that assigns types to program phrases. If the type inference algorithm verifies that a program cannot "go wrong" (i.e., is free of run-time type errors) the program is accepted; otherwise, the program is rejected. Static type checking eliminates the need to perform run-time type checking and detects many programming errors at compile-time. The cost of this efficiency and security is loss of programming flexibility, because no decidable type inference system can be both sound and complete—some programs that cannot go wrong must be rejected in any statically typed language.

Dynamically typed languages, such as Lisp and Scheme, impose no type constraints on programs and, in the worst case, perform all type-checking at run-time. This permits maximum programming flexibility at the potential cost of efficiency and security. However, in an implementation of a dynamically typed language it is beneficial to perform at least some static type checking. This regains some of the benefits of statically typed languages; how much is regained depends on the power of the type inference algorithm. Type inference systems for dynamically typed languages have been dubbed *soft typing* systems by Cartwright and Fagan [6].

Interest in inferring types for dynamically typed programs began with Reynolds [19]. Since then, numerous algorithms have been proposed, based variously on tree grammars [13, 23], *ad hoc* extensions of static type systems [6, 10, 21], abstract interpretation [2, 20], and constraint solving [11, 12]. Many of these techniques are complicated, limited in power, or both.

In this paper we present a soft typing system that is both simple and powerful. Our method is simple because we address a type inference problem directly using type inference techniques; there are no special encodings or awkward cases to consider. Our method is also powerful: our algorithm automatically infers accurate types for recursive, higher-order, dynamically typed programs with no type information supplied by the user.

The essence of our approach is to perform standard ML-style type inference, but over a much richer domain of types. Our type language (Section 3) includes function types, a least type 0, a greatest type 1, intersection, union, recursive types, and *conditional* types (see below). A program is *well-typed* if it is provable that the program is free of run-time type errors. Our type inference system reduces the problem of determining whether a program is well-typed to the satisfiability of a system of type inclusion constraints (Section 4). The inclusion constraints are solvable, which provides an effective type inference procedure (Section 5).

The most novel feature of our type language is conditional types (Section 3). With conditional types, the type of an expression e can be constrained using information about the results of run-time tests in the context surrounding e. For example, in an expression

$$\textbf{case } e_1 \textbf{ of true} : e_2, \textbf{false} : e_3$$

conditional types can express that e_2 is evaluated only in environments where e_1 is *true*, and e_3 is evaluated only in environments where e_1 is *false*. This kind of analysis usually is called *control-flow analysis*. The ability to take advantage of control-flow information in type inference is crucial to computing accurate type information in dynamically typed programs [2, 20]. Using conditional types, we eliminate the *ad hoc* steps in [2, 20] used to perform control-flow analysis.

The type inference system presented in Section 4 is quite

*Author's current address: Computer Science Division, University of California at Berkeley, Berkeley, CA 94720, email: aiken@cs.berkeley.edu

†This work was done while visiting IBM Almaden Research Center.

To appear in Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

powerful and interesting in its own right. However, since no decidable type inference system can be both sound and complete, our system rejects some programs that make no run-time type errors. In a dynamically typed language programs cannot be rejected by a type system—after all, the language itself imposes no type constraints. Following [6, 12], we take the view that explicit run-time type checks should be added to the program to make it well-typed. Section 7 describes how our system automatically adds such type checks to a program.

This paper makes three contributions. First, we believe our system infers the most accurate types of any proposed type inference system for dynamically (or statically) typed languages. Second, we show how control-flow analysis can be performed using type inference with conditional types. Third, we show that several proposals for performing program analysis of dynamically typed languages are in fact special cases of solving systems of type inclusion constraints. Thus, our approach unifies a diverse body of work on program analysis (Section 9).

The type inference system described here has been implemented and extensively tested for the functional programming language FL. Section 6 presents examples, while Section 8 discusses the implementation and presents the results of performance measurements. Proofs and some technical discussion are deferred to appendices.

2 A Programming Language

We illustrate our system using a simple dynamically typed, higher-order functional language. Our system should be extensible to imperative language features using standard techniques [22]. The programming language $\mathcal{L}$ is the lambda calculus with constructors, **let**, **case**, and patterns:

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e_1 \mid e_1\ e_2 \mid c(e_1,\ldots,e_n) \mid \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \\ & & \textbf{case}\ e\ \textbf{of}\ p_1 : e_1, \ldots, p_n : e_n \\ p & ::= & x \mid x\ \textbf{as}\ p_1 \mid c(p_1,\ldots,p_n) \end{array}$$

Giving a formal semantics to $\mathcal{L}$ is routine and we omit it. We briefly summarize the important features of the semantics. The semantic domain D of $\mathcal{L}$ satisfies the equations

$$\begin{aligned} D^+ &= (D^+ \to D) \cup \bigcup_{c \in C} c(D^+,\ldots,D^+) \\ D &= D^+ \cup \{\bot, \mathit{wrong}\} \end{aligned}$$

The set C is a set of data constructors (e.g., constants, pairs, etc.), the value $\bot$ denotes non-termination, and the value *wrong* denotes an erroneous computation.

The meaning function $\mu : \mathrm{Expr} \times \mathrm{Env} \to D$ for $\mathcal{L}$ expressions maps an expression e and an environment assigning meaning to the free variables of e to an element of D. Expressions are generally strict in $\bot$ and *wrong*; for example, $f\ \bot = \bot$, $c(\bot, x) = \bot$, etc. The exceptions are lambda abstraction ($\lambda x.\ \bot$ is a function that returns $\bot$) and **case** (see below). The choice of a strict semantics over a lazy semantics is not important; our techniques work for lazy languages as well.

In an expression **case** e **of** $p_1 : e_n, \ldots, p_n : e_n$, if e evaluates to v and v matches the "shape" of pattern p_i, then the result is e_i with the variables in p_i bound to corresponding components of v. The other branches are not evaluated. The pattern x **as** p binds x to a value that matches p. If no pattern matches a **case** analysis, the expression evaluates to *wrong*. For example, $\mu(\textbf{case false of true} : \textbf{nil}, \emptyset) = \mathit{wrong}$. We impose two standard restrictions on patterns. Patterns must be *linear* (each variable in a pattern occurs exactly once). Within a single **case**, patterns must be pairwise disjoint (so that no two patterns match the same value).

We prefer to use **case** instead of a conditional **if** because **case** is more general. The usual conditional **if** $e_1\ e_2\ e_3$ is defined as **case** e_1 **of true** : e_2, **false** : e_3.

3 Types

This section presents the syntax and semantics of our type language. The syntax of type expressions is given by the following grammar:

$$\begin{array}{rcl} \tau & ::= & \tau_1 \to \tau_2 \mid c(\tau_1,\ldots,\tau_n) \mid \alpha \mid \tau_1 \cup \tau_2 \mid \tau_1 \cap \tau_2 \mid \tau_1 ? \tau_2 \mid 0 \mid 1 \\ \sigma & ::= & \tau \mid \forall \alpha_1,\ldots,\alpha_n.\tau\ \mathit{where}\ \{\ldots, \tau_1 \subseteq \tau_2, \ldots\} \end{array}$$

Unquantified types are written $\tau, \tau_1, \tau_2, \ldots$. Types that may be quantified are written $\sigma, \sigma_1, \sigma_2, \ldots$.

For semantics we adopt the *ideal* model, in which types are certain subsets (called ideals) of the semantic domain D [15]. In the ideal model, every type τ satisfies four conditions: τ is non-empty, *wrong* $\notin \tau$, τ is *directed-closed* (closed under limits), and τ is *downward-closed*, which means that if $y \in \tau$ and if $x \le y$[1] then $x \in \tau$. Since types are sets, types are ordered by set inclusion.

In the grammar above, α is a type variable. Given an assignment ρ of types to type variables, Figure 1 extends ρ to give semantics to all type expressions. We briefly explain each case. The first three type expressions are familiar from typed functional languages. They are function types $\tau_1 \to \tau_2$ (the set of functions mapping elements of τ_1 to elements of τ_2), constructor types $c(\tau_1,\ldots,\tau_n)$ (the set of "c" data structures with components drawn from $\tau_1,\ldots,\tau_n$), and type variables. Note that we use the same name c for both a value constructor in expressions and a type constructor in types. The type expressions $\tau_1 \cup \tau_2$ and $\tau_1 \cap \tau_2$ denote set-theoretic union and intersection of types respectively.

The type 0 contains only $\bot$, the value denoting non-termination. Since types are non-empty and downward-closed, 0 is the least type: $0 \subseteq \tau$ for any type τ. The type 1 is the entire semantic domain except *wrong*. Note that 1 is the greatest type: $\tau \subseteq 1$ for any type τ. The type 1 is handy for defining the set of all values of a particular kind. For example, the set of all functions (that don't go wrong) is $1 \to 1$, the set of all **cons** pairs is $\textbf{cons}(1,1)$, and so on. The type $\textbf{cons}(1,1)$ is an example of a *monotype*—a type with no variables. For monotypes and types with no free (unquantified) variables, the type denotes the same set regardless of the choice of substitution in Figure 1. In this case we drop the substitution and treat the the type expression itself as a set.

The type $\tau_1 ? \tau_2$, read "τ_1 if τ_2", is a *conditional* type. Conditional types can express a restricted form of overloading, which is useful in giving accurate types for **case** expressions. For example, consider the function

$$\lambda y.\textbf{case}\ y\ \textbf{of true} : \textbf{zero},\ \textbf{false} : \textbf{succ}(\textbf{zero})$$

[1] The standard ordering: $\bot \le x$, $c(x_1,\ldots,x_n) \le c(y_1,\ldots,y_n)$ iff $x_i \le y_i$, and $f \le g$ iff $f(x) \le g(x)$ for all x

$$
\begin{aligned}
\rho(\tau_1 \to \tau_2) &= \{f | f(\rho(\tau_1) - \{\bot, wrong\}) \subseteq \rho(\tau_2)\} \cup \{\bot\} \\
\rho(c(\tau_1, \ldots, \tau_n)) &= \{c(t_1, \ldots, t_n) | t_i \in \rho(\tau_i) - \{\bot, wrong\}\} \cup \{\bot\} \\
\rho(\tau_1 \cup \tau_2) &= \rho(\tau_1) \cup \rho(\tau_2) \\
\rho(\tau_1 \cap \tau_2) &= \rho(\tau_1) \cap \rho(\tau_2) \\
\rho(\tau_1 ? \tau_2) &= \begin{cases} \rho(\tau_1) & \text{if } \rho(\tau_2) \neq \{\bot\} \\ \{\bot\} & \text{otherwise} \end{cases} \\
\rho(0) &= \{\bot\} \\
\rho(1) &= D - \{wrong\} \\
\\
\rho(\forall \alpha_1, \ldots, \alpha_n.\tau \ \mathit{where}\ S) &= \bigcap_{\rho' \in X} \rho'(\tau) \\
& \quad \text{where } X = Sol(S) \cap \{\rho' | \rho'(\beta) = \rho(\beta) \text{ if } \beta \notin \{\alpha_1, \ldots, \alpha_n\}\}
\end{aligned}
$$

Figure 1: Semantics of type expressions.

A fragment of the type inferred for this function by our algorithm is

$$\alpha \to (\mathbf{zero}?(\alpha \cap \mathbf{true})) \cup (\mathbf{succ}(\mathbf{zero})?(\alpha \cap \mathbf{false}))$$

Substituting **true** (resp. **false**) for α and simplifying,[2] this type has instance $\mathbf{true} \to \mathbf{zero}$ (resp. $\mathbf{false} \to \mathbf{succ}(\mathbf{zero})$). Thus, this type accurately captures the input-output dependencies of the **case** expression. The ability to constrain the types of expressions using information about run-time tests is usually called *control-flow analysis* and is very important in inferring accurate types for dynamically typed programs [20]. Conditional types were first introduced by Reynolds in his algorithm for analyzing Lisp programs [19].

Type schemes have the form $\forall \alpha_1, \ldots, \alpha_n.\tau \ \mathit{where}\ S$ where α_i is a type variable, τ is an unquantified type expression, and S is a set of *type constraints* of the form $\tau_1 \subseteq \tau_2$. The unusual aspect of our type schemes is the use of subsidiary constraints. Intuitively, the constraints are a form of bounded quantification restricting τ to instances satisfying the constraints. More formally, the *solutions* $Sol(S)$ of a system S of constraints is the set of assignments ρ of types to type variables such that $\rho(\tau_1) \subseteq \rho(\tau_2)$ holds for all constraints $\tau_1 \subseteq \tau_2$ in S. In the case where $\forall \alpha_1, \ldots, \alpha_n.\tau \ \mathit{where}\ S$ is fully quantified (i.e., there are no free type variables in τ), the meaning is the intersection $\bigcap_{\rho \in Sol(S)} \rho(\tau)$. For example, the type $\forall \alpha.\alpha \to \alpha \ \mathit{where}\ \emptyset$ is just the type of the identity function. However, $\forall \alpha.\alpha \to \alpha \ \mathit{where}\ \{\alpha \subseteq \mathbf{int}\}$ is the type of the identity on the integers. For the function using **case** defined above, the full type inferred by our algorithm is

$$\forall \alpha. \quad \alpha \to (\mathbf{zero}?(\alpha \cap \mathbf{true})) \cup (\mathbf{succ}(\mathbf{zero})?(\alpha \cap \mathbf{false})) \quad \mathit{where}\ \{\alpha \subseteq \mathbf{true} \cup \mathbf{false}\}$$

The constraint on α says the function is guaranteed to be well-defined only if applied to the constructors **true** or **false**.

Recursive types are not included in the grammar for type expressions because recursive types are definable using constraints. Let $\tau_1 = \tau_2$ stand for the pair of constraints $\tau_1 \subseteq \tau_2$ and $\tau_2 \subseteq \tau_1$. Then a constraint such as $\alpha = \mathbf{cons}(\beta, \alpha) \cup \mathbf{nil}$ has a unique solution which defines (given the usual interpretation of **cons** and **nil**) α to be all lists with elements of type β.

We assume the set of type constructors includes constants such as **zero**, **true**, **false**, and **nil**, a unary constructor **succ**, and a binary constructor **cons**. We separate **true** and **false** (resp. **nil** and **cons**) from the more conventional type **bool** (resp. **list**) in order to assign more precise types to expressions.

4 Type Inference

A type inference system for $\mathcal{L}$ is given in Figure 2. Associated with every conclusion is a set of assumptions A and a set of type constraints S. The rules prove sentences of the form "$A, S \vdash e : \tau$", which should be read "if the free variables of e have the types given by assumptions A, then e has type $\rho(\tau)$ for any solution ρ of the constraints S." That is, the conclusion holds only for solutions of S.

We briefly explain the inference rules in Figure 2. The rule [VAR] is standard: given the assumption $x : \sigma$ one can prove $x : \sigma$. The rule [STRUCT] says that the type of a construction is the construction of the component types. For nullary constructors such as **true**, the scheme in [STRUCT] simplifies to an axiom $A, S \vdash \mathbf{true} : \mathbf{true}$ (recall that a constructor name is used both in values and types). In the [APP] rule, if $\tau_3 \to \tau_4$ is a superset of the type for e_1 and τ_3 is a superset of the type for e_2, then τ_4 is a type for $e_1\, e_2$.

To simplify presentation of the [CASE] rule, we introduce two auxiliary functions for patterns. The set of all variables in pattern p is $V(p)$. The type $\overline{p}$ is the set of all values that can match the pattern p:

$$\overline{x} = 1 \qquad \overline{x \ \mathbf{as}\ p} = \overline{p} \qquad \overline{c(p_1, \ldots, p_n)} = c(\overline{p_1}, \ldots, \overline{p_n})$$

For example, the pattern $\mathbf{cons}(x, y)$ matches any **cons** pair. This fact is captured by $\overline{\mathbf{cons}(x, y)} = \mathbf{cons}(1, 1)$.

For an expression **case** e **of** $p_1 : e_1, \ldots, p_n : e_n$, the rule [CASE] takes as hypotheses a type τ for e, a type τ_i for each e_i, and a type τ_i' for each p_i. The type τ_i appears in the conclusion of the rule in the disjunct $\tau_i ? (\tau \cap \overline{p_i})$. Thus, τ_i is included in the result type if there is something in τ matching p_i. Otherwise, if $\tau \cap \overline{p_i} = 0$ then this branch

[2] $\mathbf{true} \to (\mathbf{zero}?(\mathbf{true} \cap \mathbf{true})) \cup (\mathbf{succ}(\mathbf{zero})?(\mathbf{true} \cap \mathbf{false})) = \mathbf{true} \to (\mathbf{zero}?\mathbf{true}) \cup (\mathbf{succ}(\mathbf{zero})?0) = \mathbf{true} \to \mathbf{zero} \cup 0 = \mathbf{true} \to \mathbf{zero}$

$$\frac{}{A \cup \{x : \sigma\}, S \vdash x : \sigma} \quad [VAR]$$

$$\frac{A, S \vdash e_i : \tau_i \quad 1 \leq i \leq n}{A, S \vdash c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n)} \quad [STRUCT]$$

$$\frac{A \cup \{x : \tau_1\}, S \vdash e : \tau_2}{A, S \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \quad [ABS]$$

$$\frac{A, S \vdash e_1 : \tau_1,\ e_2 : \tau_2}{A, S \cup \{\tau_2 \subseteq \tau_3,\ \tau_1 \subseteq \tau_3 \rightarrow \tau_4\} \vdash e_1\ e_2 : \tau_4} \quad [APP]$$

$$\frac{A, S \vdash x : \tau_1,\ p : \tau_2}{A, S \vdash x\ \textbf{as}\ p : \tau_1 \cap \tau_2} \quad [AS]$$

$$\frac{\begin{array}{l} A, S \vdash e : \tau \\ A \cup \{x : \tau_x | x \in V(p_1)\}, S \vdash e_1 : \tau_1,\ p_1 : \tau_1' \\ \vdots \\ A \cup \{x : \tau_x | x \in V(p_n)\}, S \vdash e_n : \tau_n,\ p_n : \tau_n' \end{array}}{A, S \cup \{\tau \subseteq \bigcup_{1 \leq i \leq n} \tau_i'\} \vdash \textbf{case}\ e\ \textbf{of}\ p_1 : e_1, \ldots, p_n : e_n\ : \bigcup_{1 \leq i \leq n} \tau_i ? (\tau \cap \overline{p_i})} \quad [CASE]$$

$$\frac{A, S \vdash e_1 : \sigma \quad A \cup \{x : \sigma\}, S \vdash e_2 : \tau}{A, S \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \quad [LET]$$

$$\frac{A, S \vdash e : \tau \text{ and } Sol(S) \neq \emptyset \text{ and } \alpha_1, \ldots, \alpha_n \text{ not free in } A}{A, \emptyset \vdash e : \forall \alpha_1, \ldots, \alpha_n.\tau\ \ where\ S} \quad [GEN]$$

$$\frac{A, S \vdash e : \forall \alpha_1, \ldots, \alpha_n.\tau\ \ where\ S'}{A, S \cup S'[\tau_i/\alpha_i] \vdash e : \tau[\tau_i/\alpha_i]} \quad [INST]$$

Figure 2: Type inference rules.

$$\dfrac{\dfrac{\{x:\alpha, y:1\}, \emptyset \vdash x : \alpha}{\{x:\alpha\}, \emptyset \vdash \lambda y.x : 1 \to \alpha}}{\vdash \lambda x.\lambda y.x : \alpha \to 1 \to \alpha}$$

(a) The K combinator.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{y:\mathbf{true}\}, \emptyset \vdash y : \mathbf{true}} \quad \dfrac{}{\{y:\mathbf{true}\}, \emptyset \vdash \mathbf{zero} : \mathbf{zero}} \quad \dfrac{}{\{y:\mathbf{true}\}, \emptyset \vdash \mathbf{succ(zero)} : \mathbf{succ(zero)}}}{\{y:\mathbf{true}\}, \emptyset \vdash \mathbf{case}\ y\ \mathbf{of\ true} : \mathbf{zero},\ \mathbf{false} : \mathbf{succ(zero)}\ : \mathbf{zero}} \quad \dfrac{}{\vdash \mathbf{true} : \mathbf{true}}}{\vdash \lambda y.\mathbf{case}\ y\ \mathbf{of\ true} : \mathbf{zero},\ \mathbf{false} : \mathbf{succ(zero)} : \mathbf{true} \to \mathbf{zero}}}{\vdash (\lambda y.\mathbf{case}\ y\ \mathbf{of\ true} : \mathbf{zero},\ \mathbf{false} : \mathbf{succ(zero)})\ \mathbf{true} : \mathbf{zero}}$$

(b) An example using **case**.

Figure 3: Examples of well-typed expressions.

of the case cannot be taken and $\tau_i?(\tau \cap \overline{p_i}) = \tau_i?0 = 0$. The constraint $\tau \subseteq \bigcup_{1 \le i \le n} \tau_i'$ serves two purposes. First, it ensures that the set of **case** branches is an exhaustive analysis of the type τ. Second, this constraint propagates type information about e to the types of the variables in the e_i.

Finally, the rules [LET], [GEN], and [INST] are standard except for the use of subsidiary constraints in [GEN] and [INST]. The following lemma shows that the inference system is sound.

Lemma 4.1 (Soundness) Let $A, S \vdash e : \sigma$, let ρ be any solution of the constraints S, and let E be an environment for the free variables of e such that if $E(x) = v_x$ and $x : \sigma_x \in A$, then $v_x \in \rho(\sigma_x)$. Then $\mu(e, E) \in \rho(\sigma)$.

Definition 4.2 A *program* is an $\mathcal{L}$ expression with no free variables.

Corollary 4.3 Let e be program and let $\emptyset, \emptyset \vdash e : \sigma$. Then $\mu(e, \emptyset) \in \sigma$.

The sentence $\emptyset, \emptyset \vdash e : \sigma$ is abbreviated $\vdash e : \sigma$. Recall that *wrong* $\notin \tau$ for any type τ (Section 3). The following corollary is immediate.

Corollary 4.4 If $\vdash e : \sigma$ then $\mu(e, \emptyset) \neq$ *wrong*.

A program e is *well-typed* if $\vdash e : \sigma$. Figure 3a gives a proof that the K combinator is well-typed. Figure 3b is a more complex example involving a **case** expression. In Figure 3b, some types have been simplified. For instance, by rule [CASE] the first conclusion is $\mathbf{zero}?(\mathbf{true} \cap \mathbf{true}) \cup \mathbf{succ(zero)}?(\mathbf{false} \cap \mathbf{true})$, but this is $\mathbf{zero}?\mathbf{true} \cup \mathbf{succ(zero)}?0 = \mathbf{zero}$. In addition, we have elided the constraints, which are all vacuously true in this proof. For instance, by rule [CASE] there is a constraint $\mathbf{true} \subseteq \mathbf{true} \cup \mathbf{false}$ in the first conclusion, but this always holds, and so has the same solutions as the empty set of constraints.

The inference system in Figure 2 is a very powerful static type system and may be of interest in its own right. In our application, the reason for having such a powerful system is to make the set of well-typed programs as large as possible, because any program that is well-typed requires no run-time type checking. To make a convincing case for using this type inference system with a dynamically typed language, there are still three things that need to be explained. In the rest of this section, we show that the set of well-typed programs is in fact very large. This shows that many programs pass the type checker without modification. In Section 5, we show that the type constraints can be solved—this shows that it is decidable whether or not a program is well-typed. Finally, in Section 7 we explain how dynamic type checks can be inserted into ill-typed programs to make them well-typed; this guarantees that every program can be modified to pass the type checker.

It is important to know the "size" of the set of well-typed programs. If the set is small, than the type inference system is not of much practical interest. We do not have an exact characterization of the set of well-typed programs, but the following two lemmas show that the set is very large.

Lemma 4.5 A *lambda term* is a program without constructors or **case**. Every lambda term is well-typed.

The Hindley/Milner type inference system [16] is used in most functional languages. We write $\vdash_{HM} e : \sigma$ if e has type σ in the Hindley/Milner system. Lemma 4.6 shows that all programs well-typed in the Hindley/Milner system are also well-typed in our system.

Lemma 4.6 If $\vdash_{HM} e : \sigma$ then $\vdash e : \sigma$.

5 Computing Types

In this section we discuss an algorithm that derives types in the logic given in Figure 2. We do this in two steps: first, we prove that expressions have *minimal* types in this logic. We then show that the minimal type is computable.

Definition 5.1 A program e has *minimal type* σ iff $\vdash e : \sigma$ and for any σ', if $\vdash e : \sigma'$ then $\sigma \subseteq \sigma'$.

Minimal types are related to, but not the same as, *principal* types [7]. Principal types are defined syntactically, with the usual definition being that σ is a principal type for e if $\vdash e : \sigma$ and if for any other type σ' such that $\vdash e : \sigma'$, the type σ' is a substitution instance of σ. In the Hindley/Milner system the two notions coincide, so that a type is principal if and only if it is minimal in the ideal model. Thus, in this instance minimal types are simply the semantic counterpart of principal types. In our system, programs have minimal types but may not have principal types (e.g., because many type expressions denote the same type).

A minimal type is the smallest derivable type in the semantic model. To prove that programs have minimal types, we introduce the notion of a most general type derivation.

$$
\begin{array}{l}
\textbf{let } Y = \lambda u.(\lambda x.(\lambda w.(u\ (x\ x))\ w)\ (\lambda w.(u\ (x\ x))\ w)\ \textbf{in} \\
\quad \textbf{let } last = Y\ \lambda f.\lambda x.\textbf{case } x \textbf{ of} \\
\qquad\qquad\qquad\qquad \textbf{cons}(y, \textbf{nil}) : y \\
\qquad\qquad\qquad\qquad \textbf{cons}(u, v \textbf{ as } cons(a, b)) : f\ v \\
\quad \textbf{in} \\
\qquad \textbf{let } a = last\ \textbf{cons}(\textbf{zero}, \textbf{cons}(\textbf{zero}, \textbf{cons}(\textbf{zero}, \textbf{nil}))) \textbf{ in} \\
\qquad\quad last\ \textbf{cons}(\textbf{true}, \textbf{cons}(\textbf{false}, \textbf{cons}(a, \textbf{nil})))
\end{array}
$$

Figure 4: Taking the last element of a list.

Definition 5.2 A *most general* type derivation satisfies:

1. Every type assumption has the form $x : \alpha$ where α is a distinct type variable.
2. In every use of the [APP] rule the types τ_3 and τ_4 are fresh type variables.
3. In every $\textbf{let } x = e \textbf{ in } e'$, the [GEN] rule is applied once to quantify as many type variables as possible in the type of e.
4. The [GEN] rule is applied in the last step of the entire type derivation to quantify over all type variables.
5. The [INST] rule is applied immediately after uses of the rule [VAR] to eliminate any quantified variables. Fresh type variables are substituted for the quantified variables.
6. The [GEN] and [INST] rules are applied nowhere else in the derivation.

It is easy to check that the restrictions in Definition 5.2 specify a unique derivation up to renaming of type variables. Lemma 5.3 shows that the minimal type of a program is the one generated by a most general derivation.

Lemma 5.3 If $\vdash e : \sigma$ by a most general derivation, then σ is the minimal type for e.

To obtain an effective type inference procedure, the only missing link is an algorithm to solve the type inclusion constraints generated in a most general derivation. We make use of the following theorem.

Theorem 5.4 It is decidable whether systems of *proper* constraints have solutions. Furthermore, all solutions can be exhibited [3].

The constraints generated by a most general derivation are proper, with an extension to handle conditional types, which are not treated in [3]. A definition of proper constraints and the extension for conditional types are given in Appendix B.

6 Examples

In this section we present two examples illustrating the power of our type inference system. The first example is an implementation of the Lisp *car* function, which returns the first component of a **cons** when applied to a **cons**, **nil** when applied to **nil**, and goes wrong for any other argument.

$$
\begin{array}{l}
\textbf{let } car = \lambda x.\textbf{case } x \textbf{ of } \textbf{cons}(y, z) : y,\ \ \textbf{nil} : \textbf{nil } \textbf{in} \\
\quad car\ \textbf{cons}(\textbf{zero}, \textbf{nil})
\end{array}
$$

Our system reports that $car\ \textbf{cons}(\textbf{zero}, \textbf{nil}) : \textbf{zero}$. The interesting aspect of this example is the type of *car*, which illustrates the role of conditional types in our system. The type inferred for *car* is

$$
\begin{array}{c}
\forall \alpha, \beta, \gamma.\quad \alpha \to (\beta?(\alpha \cap \textbf{cons}(1, 1))) \cup (\textbf{nil}?(\alpha \cap \textbf{nil})) \\
where\ \{\alpha \subseteq \textbf{cons}(\beta, \gamma) \cup \textbf{nil}\}
\end{array}
$$

If we let $\alpha = \textbf{cons}(\textbf{zero}, 1)$, $\beta = \textbf{zero}$, and $\gamma = 1$, then this type has instance $\textbf{cons}(\textbf{zero}, 1) \to \textbf{zero}$. Thus, the application has type **zero**; this is also the minimal type. In addition to handling the distinction between **cons** and **nil** accurately, the inferred type for *car* is completely accurate even for heterogeneous lists, since only the type of the head of the list is relevant. For example, our system also reports that $car\ \textbf{cons}(\textbf{zero}, \textbf{cons}(\textbf{true}, \textbf{nil})) : \textbf{zero}$.

A more substantial example is given in Figure 4, which defines a recursive function *last* that selects the last element of a list. Note that a Y combinator is used for recursion because $\mathcal{L}$ has no special syntax for declaring recursive functions. The particular version of Y used here is one appropriate for a strict language such as $\mathcal{L}$. The function *last* is used polymorphically in two places: once with a list of **zero** and once with a heterogeneous list consisting of some booleans followed by **zero**.

This expression illustrates some of the difficulties of performing automatic type inference in a dynamically typed language such as Lisp or Scheme. The program contains no type declarations—everything must be inferred. The program uses higher-order, recursive, polymorphic functions. The function *last* works for any heterogeneous list of at least length one; for any other input value, it goes wrong. Control-flow analysis is required to make distinctions between what *last* does with the last element of the list and all other elements of the list. The expression in Figure 4 means **zero**. Our algorithm infers that this expression has type **zero**. Therefore, it is well-typed and no run-time type checking is required.

The key to proving this expression is well-typed is the type for *last*. The minimal type of *last* computed by our system is:

$$
\forall \alpha.X \to \alpha\ \ where\ \{X = \textbf{cons}(1, X) \cup \textbf{cons}(\alpha, \textbf{nil})\}
$$

In this example we have rewritten the type of *last* produced by our system to an equivalent, but more readable, form. The type X is the set of all lists of length at least one ending in α. Because of the polymorphic **let**, the two uses of *last* are typed separately using distinct instances of the type above. In the first instance *last* has type

$$
X' \to \textbf{zero}\quad where\ \{X' = \textbf{cons}(1, X') \cup \textbf{cons}(\textbf{zero}, \textbf{nil})\}
$$

The result type **zero** is the type of the variable a. Since the second application of *last* is to a list with last element a, this instance is also assigned the type $X' \to \mathbf{zero}$. The type of the application (and of the whole expression) is then **zero**.

7 Inserting Dynamic Type Checks

Recall that an expression e is well-typed if $\vdash e : \sigma$. If e is ill-typed then it cannot be said whether or not e makes a run-time type error. In a dynamically typed language, it is not reasonable to reject ill-typed programs, because no type constraints are imposed on programs. Instead, it is necessary to add dynamic type checks to the program to make it well-typed. To make an ill-typed program well-typed we insert functions called *narrowers* [6] in certain places. A narrower performs a run-time check and returns a value indicating whether or not the check succeeded. A narrower $Check_X$ is defined for every monotype X. The meaning of $Check_X$ is

$$Check_X\ v = \begin{cases} v & \text{if } v \in X \\ \bot & \text{otherwise} \end{cases}$$

$Check_X$ is the identity on X and $\bot$ elsewhere. Thus, $Check_X$ "narrows" the set of possible values of the input to X. The type of $Check_X$ is $\forall\alpha.\alpha \to \alpha \cap X$.

From Figure 2 and Definition 5.2 there are three kinds of constraints in a most general derivation:

$$\begin{array}{lcll} \tau & \subseteq & \alpha & \text{from [APP]} \quad (1) \\ \tau & \subseteq & \alpha \to \beta & \text{from [APP]} \quad (2) \\ \tau & \subseteq & \bigcup_i \tau_i' & \text{from [CASE]} \quad (3) \end{array}$$

In a most-general derivation, constraints of form (1) are satisfied by the assignment $\alpha = 1$. Thus, only constraints (2) and (3) may be inconsistent and result in an ill-typing. Constraint (2) guarantees that e_1 is a function in an application $e_1\ e_2$. Constraint (3) guarantees that all possible values of e are covered by the case analysis in **case** e **of** ....

The following lemma shows that by inserting narrowers in all applications and **case** expressions, any program can be made well-typed. In other words, there is at least one way to add enough dynamic type checking to guarantee that a program makes no run-time type errors.

Lemma 7.1 Let e be a program and let e' be e modified as follows:

1. Replace every application $e_1\ e_2$ by $(Check_{1\to 1}\ e_1)\ e_2$.
2. Replace every **case** e **of** $p_1 : e_1, \ldots, p_n : e_n$ by **case** $(Check_X\ e)$ **of** $p_1 : e_1, \ldots, p_n : e_n$ where $X = \bigcup_{1 \le i \le n} \overline{p_i}$.

Then e' is well-typed.

Lemma 7.1 gives one very conservative way of inserting dynamic checks. To minimize the run-time overhead of dynamic type checking, our system attempts to minimize the number of checks inserted. This is done performing some extra bookkeeping in constraint solving, so that it is possible to tell which constraints of forms (2) and (3) are violated in an ill-typed program. The following trivial example illustrates the idea. The type derivation for the ill-typed application (**true false**) generates the constraints:

$$\mathbf{false} \subseteq \alpha \qquad \mathbf{true} \subseteq \alpha \to \beta$$

The constraints are inconsistent since $\mathbf{true} \not\subseteq \alpha \to \beta$. Therefore, a dynamic type check is required on the application. The expression is modified to $((Check_{1\to 1}\ \mathbf{true})\ \mathbf{false})$. It is easy to verify that the modified program is well-typed and that its minimal type is 0, thereby proving that its meaning is $\bot$.

The rest of this section describes how inconsistent constraints are detected by the implementation; this is the "extra bookkeeping" mentioned above. In fact, checking which constraints of forms (2) and (3) are violated is done by solving an extended system of constraints. The extended constraints always have at least one solution, so the constraint solver is guaranteed to terminate successfully. By examining the solutions of this extended system of constraints, it is possible to determine conservatively which constraints of forms (2) and (3) are violated in the original constraint system.

The definition of the extended constraints requires the introduction of the type $\neg X$, the complement of type X.

Definition 7.2 $\neg X$ is the largest type such that $\neg X \cap X = 0$.

The type $\neg X$ is unique, so $\neg X$ is well-defined. Also, constraints involving negated types can be solved; for details see [3]. As an example, the type $\neg(1 \to 1)$ is the type of all non-functions, which is $\bigcup_{c \in C} c(1, \ldots, 1)$ where C is the set of all data constructors.

Consider the application constraint $\mathbf{true} \subseteq \alpha \to \beta$ from the example above. This constraint is inconsistent (has no solutions) because the right-hand side can contain at most the set of all functions. To modify this constraint so that it always has a solution, an *error term* is added to the right-hand side to include those values that result in run-time errors. The modified constraint is

$$\mathbf{true} \subseteq (\alpha \to \beta) \cup (\gamma \cap \neg(1 \to 1))$$

This constraint is satisfiable, since if $\alpha = \beta = \gamma = 1$ then the right-hand is 1. In addition, by setting the error variable $\gamma = 0$ the error term drops out and the original constraint is recovered. The intuition is that the error term allows for all (and only) the values that may cause a run-time error. The error variable γ makes it possible to distinguish between solutions where there are no run-time errors ($\gamma = 0$) and solutions that may admit run-time errors ($\gamma \neq 0$). For example, the constraint above implies that $\mathbf{true} \subseteq \gamma$. Thus, there is no solution where $\gamma = 0$, which implies that the original constraint is inconsistent.

In general, the extended constraints include constraints of form (1) and modifications of forms (2) and (3):

$$\begin{array}{lcll} \tau & \subseteq & \alpha \to \beta \cup (\gamma \cap \neg(1 \to 1)) & (2') \\ \tau & \subseteq & (\bigcup_i \tau_i') \ \cup \ (\gamma \cap \neg \bigcup_i \overline{\tau_i'}) & (3') \end{array}$$

In each case the error term covers exactly those values that may cause a run-time error. These constraints are proper and can be solved using the algorithm in [3]. In addition, these constraints always have a solution (e.g., let all variables be 1; then the right-hand side of every constraint is 1).

Let S be an extended system of constraints generated from a program e. After solving the constraints, each error variable γ is considered in turn. If the system $S \cup \{\gamma = 0\}$ has a solution, then the constraint $\gamma = 0$ is added to S. In this case, the original constraint (without the error term)

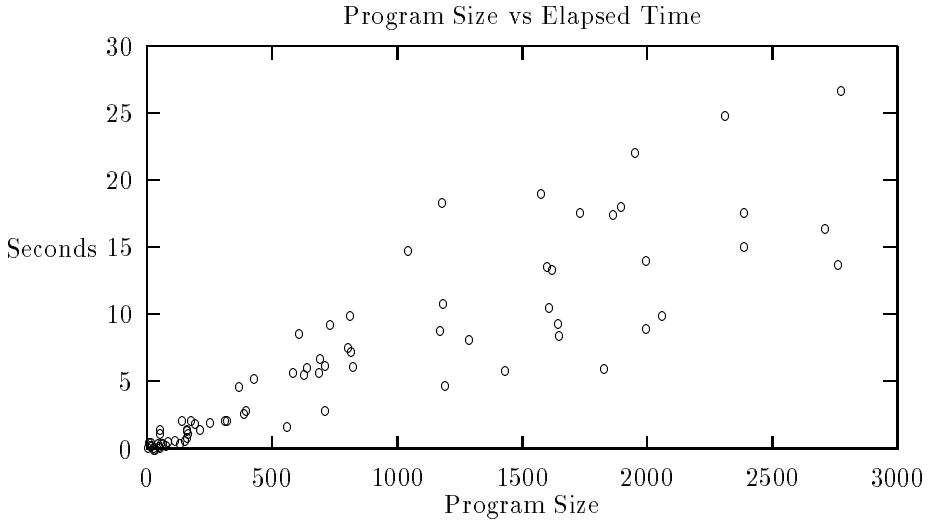

Figure 5: Performance results.

must be satisfiable, since setting $\gamma = 0$ in a constraint with an error term recovers the original constraint. It is easy to show that in this case, no run-time check is required for the program phrase of e associated with the constraint. If there is no solution of the system $S \cup \{\gamma = 0\}$, then the appropriate run-time check is added to e and S is not changed. Note that the case where all error variables are replaced by 0 corresponds to a well-typed program.

8 Implementation

There are two implementations of the type inference algorithm presented here: one for $\mathcal{L}$ and one for FL [5], a dynamically typed, higher-order functional language based on Backus' FP [4]. The implementation of type inference for $\mathcal{L}$ is small (about 100 lines of Lisp code). The implementation for FL is considerably larger (about 1000 lines of Lisp code), reflecting the increased complexity of FL, in which it is necessary to deal with exceptions, side-effects from I/O, and a rich set of primitive functions. Both implementations are functional programs built on top of an implementation of a constraint solver for type inclusion constraints.

The implementation for FL has been extensively tested on a diverse suite of about 80 FL programs, which range from small utilities (e.g., sort) to modules of several hundred lines of code (e.g., a general I/O package). Figure 5 gives performance results for the test suite. Program "size" in this figure is the number of nodes in the program's parse tree. This is a more relevant measure than lines of code, since the number of parse tree nodes corresponds to the number of steps in a type derivation. For the test suite programs, the number of lines of code is anywhere between a factor of 2 and 10 less than the number of parse tree nodes. For example, *escher* implements a simple graphics language; *escher* has 278 lines of FL code, which translates into 2770 parse tree nodes. All experiments were done on an unloaded IBM RS/6000 running Lucid Common Lisp. The times in Figure 5 are total elapsed times.

The implemented algorithm for solving systems of type constraints requires exponential time in the worst case. From Figure 5, however, it is clear that in practice the complexity does not grow rapidly for realistic-sized modules, and the absolute speed is fast enough to be useful. Memory requirements are moderate, with a maximum of 5MB allocated for any example in the test suite. We have not measured the maximum amount of live data, but it is certainly much smaller than 5MB.

This level of performance is not trivial to achieve—a naive implementation is unusable even for small programs. We have relied heavily on a number of simple optimizations that improve the performance of the constraint solver by orders of magnitude [1]. In the current implementation, only about 10% of the time is spent solving constraints. The majority of the time is spent simplifying the representation of type expressions and determining where dynamic type checks should be inserted. Further performance improvements should be possible, as this portion of the system has not been tuned extensively.

9 Related Work

This section compares our work with a wide variety of related work. The coverage of each proposal is necessarily brief.

9.1 Type Inference Systems

A number of type inference systems for dynamically-typed languages have been proposed. Gomard's system adds to the Hindley/Milner type system an *undefined* type [10] and then uses a minor variation of the Hindley/Milner type assignment algorithm. The type *undefined* represents a value for which nothing is statically known, which is the same role

played by the type 1 in our system. Without other type operators, the types assigned by Gomard's system are not very precise, and as a result many dynamic type checks have to be added to make programs well-typed.

Another generalization of the Hindley/Milner system is *partial types*. In partial types, the type 1 is added to the usual Hindley/Milner types, but there is a different type assignment algorithm [14]. Partial types have the same limited expressive power as Gomard's system, and again the types assigned are not very precise.

Closest in spirit to our own work is *soft typing* [6]. In [6], Cartwright and Fagan set out criteria that they feel any typing algorithm for dynamically typed languages should meet and provide an algorithm meeting those criteria. The typing algorithm generates type constraints that must be solved. The constraints are not solved directly; they are first encoded in a special representation in which circular unification is used to obtain representations of solutions, which are then decoded back to types.

We feel our approach also meets the criteria in [6] for a soft typing system, while providing a simpler formalism and a more accurate type assignment algorithm. Our approach is simpler because we deal directly with the type constraints, without any intermediate encoding. With respect to accuracy, the algorithm in [6] does include union types, function types, and parametric polymorphism. It does not have intersection types or conditional types, and unions types are restricted to be *discriminative* (disjuncts in unions must have distinct outermost constructors).

At the present time we do not know how our method compares to that of [6] in practice. In future work we hope to make some empirical measurements of the strengths and weaknesses of both systems.

Outside the realm of dynamically typed languages, Fuh and Mishra [9] and Mitchell [18] have given algorithms for type inference with subtypes. Neither approach incorporates union types, parametric polymorphism, or conditional types. The subtyping algorithm of [17] has limited union and intersection types, but no function types. Also related to our work are the *refinement types* of Freeman and Pfenning [8]. Refinement types include union and intersection types, but not conditional types.

For each of the systems listed above, it is immediately apparent that our type language is more expressive, and with the exceptions of [6, 8] it is relatively easy to prove formally that our algorithm infers types that are at least as accurate. We believe such a proof is also possible with respect to the algorithm in [6], although a formal proof would be tedious to write down. Currently we do not know the exact relationship of our system to that of [8].

The greatest qualitative jump in accuracy in our system comes from the ability to encode control-flow analysis of **case** expressions; using conditional types, we are able to constrain the type of the branch of a **case** to reflect the possible values that could match the pattern. This ability is crucial to giving accurate types to many programs and it is not found in any of the systems above.

9.2 Constraints over Regular Trees

Several analysis techniques for dynamically typed languages have been based on solving equations over sets of regular trees. Reynolds proposes a method for analyzing Lisp programs [19]. This is the first and only other use of conditional types of which we are aware. Jones and Muchnick propose a different analysis system based on solving equations over sets of regular trees [13], which is used not only to eliminate dynamic type checks but also to reduce reference counting. Recently Wang and Hilfinger have proposed an analysis method based on tree grammars [23]. Since a grammar

$$X ::= X_1 \mid \ldots \mid X_n$$

is, by a small twist of perspective, a constraint

$$X = X_1 \cup \ldots \cup X_n$$

this algorithm also falls into the general class of constraint systems over regular trees.

The common weakness of these approaches is that they are inherently first-order (no function types) and monomorphic. Furthermore, none of these techniques take advantage of control-flow information. One feature found in [13, 19] is the use of *projections* $c^{-1}(c(X,Y)) = X$ to model selector functions. We do not have projections in our type language, but a selector can be assigned a function type $c(X,Y) \to X$ instead.

9.3 Abstract Interpretation

We are aware of two systems based on abstract interpretation that have been implemented. Shivers proposes a *type recovery* system for Scheme [20]. The algorithm is a classical abstract interpretation with a number of features designed specifically for type inference in dynamically typed programs. Most notably the system includes a mechanism for constraining the types of the branches of a conditional using information about the predicate.

The system of Aiken and Murphy is a predecessor to the one presented here. In [2], an analysis algorithm based on abstract interpretation combined with constraint solving over sets of regular trees is described. Essentially, the abstract interpretation is used to generate constraints that must be solved. The constraint language has no function types, intersection types are restricted, and there are no conditional types. Like Shiver's algorithm, this algorithm performs an *ad hoc* analysis to constrain the types of the branches of a conditional using the type of the predicate.

Based on several years experience with the implementation of [2], we have come to the conclusion that there are two serious problems with techniques such as these. First, in these works (as in most work on abstract interpretation) there is an implicit assumption that the entire program is available for compilation at once, which is unrealistic. Our type inference algorithm, on the other hand, is a compositional, bottom-up algorithm. This makes it amenable to use in an environment that supports separately compiled modules.

Second, while it is easy to prove that these techniques are correct, it is difficult to prove anything useful about the quality of the information the algorithms compute. Thus, it is very difficult for a programmer to predict for which programs the analysis performs well. Removing dynamic type checks makes enough difference in performance that programmers need a chance to understand the type inference system. Our algorithm is sufficiently declarative that we can give at least some guidance (Lemmas 4.5 and 4.6) to programmers who need to write efficient code.

9.4 Dynamic Typing

Henglein proposes an algorithm for *dynamic typing* that removes both dynamic type checks and dynamic tags [12]. Since our algorithm does not deal with type tags (in fact, we assume all values are tagged) our techniques do not subsume those in [12]. On the other hand, our algorithm removes more dynamic type checks than the dynamic typing algorithm, which is monomorphic, has no intersection or union types, and performs no control-flow analysis. Thus, the two algorithms are incomparable and to some extent aimed at different problems.

References

[1] Aiken, A., and Murphy, B. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture* (Aug. 1991), pp. 427–447.

[2] Aiken, A., and Murphy, B. Static type inference in a dynamically typed language. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1991), pp. 279–290.

[3] Aiken, A., and Wimmers, E. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark, June 1993), pp. 31–41.

[4] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[5] Backus, J., Williams, J. H., Wimmers, E. L., Lucas, P., and Aiken, A. The FL language manual parts 1 and 2. Tech. Rep. RJ 7100 (67163), IBM, 1989.

[6] Cartwright, R., and Fagan, M. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), pp. 278–292.

[7] Damas, L., and Milner, R. Principle type-schemes for functional programs. In *Ninth Annual ACM Symposium on Principles of Programming Languages* (1982), pp. 207–212.

[8] Freeman, T., and Pfenning, F. Refinement types for ML. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (June 1991), ACM Press, pp. 268–277.

[9] Fuh, Y., and Mishra, P. Type inference with subtypes. In *Proceedings of the 1988 European Symposium on Programming* (1988), pp. 94–114.

[10] Gomard, C. Partial type inference for untyped functional programs (extended abstract). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming* (1990), pp. 282–287.

[11] Heintze, N. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, 1992.

[12] Henglein, F. Dynamic typing. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (July 1992), pp. 205–215.

[13] Jones, N. D., and Muchnick, S. S. Flow analysis and optimization of LISP-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1979), pp. 244–256.

[14] Kozen, D., Palsberg, J., and Schwartzbach, M. I. Efficient inference of partial types. In *Foundations of Computer Science* (Oct. 1992), pp. 363–371.

[15] MacQueen, D., Plotkin, G., and Sethi, R. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages* (Jan. 1984), pp. 165–174.

[16] Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17* (1978), 348–375.

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

[18] Mitchell, J. Coercion and type inference (summary). In *Eleventh Annual ACM Symposium on Principles of Programming Languages* (Jan. 1984), pp. 175–185.

[19] Reynolds, J. C. *Automatic Computation of Data Set Definitions*. Information Processing 68. North-Holland, 1969, pp. 456–461.

[20] Shivers, O. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (June 1988), pp. 164–174.

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

[22] Tofte, M. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1987.

[23] Wang, E., and Hilfinger, P. N. Analysis of recursive types in Lisp-like languages. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (June 1992), pp. 216–225.

A Proofs

Proof: [of Lemma 4.1] This is an easy induction on the structure of the proof that $A, S \vdash e : \sigma$. □

Proof: [of Lemma 4.5] For lambda terms, type constraints are introduced only by the [APP] rule. From Figure 2, the constraints are

$$\begin{aligned} \tau_2 &\subseteq \tau_3 \\ \tau_1 &\subseteq \tau_3 \to \tau_4 \end{aligned}$$

Let $\tau_3 = \tau_4 = x$ where x is the solution of the equation $x = x \to x$. Let e be a lambda term. By induction on the

structure of the type derivation for e it is easy to check that this assignment satisfies the constraints for e. Therefore e is well-typed. □

Proof: [of Lemma 4.6] The usual Hindley/Milner rules can be expressed in our notation. Except for the [APP] and [CASE] rules, the rules in Figure 2 either are Hindley/Milner rules ([VAR], [STRUCT], [ABS], and [LET]) or differ only superficially ([GEN] and [INST]). The Hindley/Milner [APP] rule is:

$$\frac{A, S \vdash e_1 : \tau_1,\ e_2 : \tau_2}{A, S \cup \{\tau_2 = \tau_3,\ \tau_1 = \tau_3 \rightarrow \tau_4\} \vdash e_1\ e_2 : \tau_4}$$

The Hindley/Milner [CASE] rule is:

$$\frac{\begin{array}{c} A, S \vdash e : \tau \\ A \cup \{x : \tau_x | x \in V(p_1)\}, S \vdash e_1 : \tau_1, p_1 : \tau_1' \\ \vdots \\ A \cup \{x : \tau_x | x \in V(p_n)\}, S \vdash e_n : \tau_n, p_n : \tau_n' \end{array}}{\begin{array}{c} A, S \cup \{\tau = \tau_1' = \ldots = \tau_n', \tau_1 = \ldots = \tau_n\} \\ \vdash \textbf{case}\ e\ \textbf{of}\ p_1 : e_1, \ldots, p_n : e_n : \tau_1 \end{array}}$$

Since equality is a special case of containment, it is clear that the Hindley/Milner [APP] rule is a special case of the rule in Figure 2. Similarly, the constraints in the Hindley/Milner [CASE] rule are a restriction of the constraints in the [CASE] rule in Figure 2. Therefore, if $\vdash_{HM} e : \tau$ then $\vdash e : \tau$. □

Proof: [of Lemma 5.3] The type σ generated by a most general derivation has the form $\forall \alpha_1, \ldots, \alpha_n.\tau\ \mathit{where}\ S$. It is easy to show that for any other typing $\vdash e : \sigma'$, σ' is equal to $\forall \alpha_1, \ldots, \alpha_n.\tau\ \mathit{where}\ S'$ such that $Sol(S') \subseteq Sol(S)$. Since the meaning of the quantified types is the intersection of the meaning of τ in all solutions of the constraints (Section 3), it follows that $\sigma \subseteq \sigma'$. □

Proof: [of Lemma 7.1] Let $\rho(\alpha) = 1$ for all variables α. To show that e' is well-typed, it suffices to show that ρ is a solution for the type constraints in a most general derivation for e'.

Consider uses of rule [APP]. Every application in e' has the form $(Check_{1 \rightarrow 1}\ e_1)\ e_2$. For constraints of form (1), we have $\rho(\tau) \subseteq 1 = \rho(\alpha)$. Now consider constraints of form (2). If $e_1 : \tau$, then the type of $Check_{1 \rightarrow 1}$ guarantees that the type of $Check_{1 \rightarrow 1}\ e_1$ has the form $\tau \cap (1 \rightarrow 1)$. It follows that

$$\rho(\tau \cap (1 \rightarrow 1)) \subseteq 1 \rightarrow 1 = \rho(\alpha \rightarrow \beta)$$

so the type constraint is satisfied.

Consider uses of rule [CASE]. In e', every **case** expression has the form $\textbf{case}\ Check_X(e)\ \textbf{of}\ p_1 : e_1, \ldots, p_n : e_n$ where $X = \bigcup_{1 \leq i \leq n} \overline{p_i}$. If $e : \tau$, then the type of $Check_X(e)$ is $\tau \cap \bigcup_{1 \leq i \leq n} \overline{p_i}$. It is also easy to check that in a most general derivation, if $p_i : \tau_i'$ then $\rho(\tau_i') = \overline{p_i}$. Now we have that

$$\rho(\tau \cap \bigcup_{1 \leq i \leq n} \overline{p_i}) \subseteq \bigcup_{1 \leq i \leq n} \overline{p_i} = \rho(\bigcup_{1 \leq i \leq n} \tau_i')$$

and so the type constraint is satisfied. □

B Proper Constraints

We briefly explain the main result of [3] and then present the extension to conditional types. A *proper type expression* is either an R (for right) or an L (for left) type as defined by the following grammar:

$$\begin{array}{rcl} L & ::= & 0 \mid 1 \mid \alpha \mid c(L_1, \ldots, L_n) \mid R \rightarrow L \mid L_1 \cap L_2 \mid L_1 \cup L_2 \\ R & ::= & 0 \mid 1 \mid \alpha \mid c(R_1, \ldots, R_n) \mid L \rightarrow R \mid R_1 \cap R_2 \mid R_1 \cup R_2 \end{array}$$

The following restrictions are placed on R and L types. In an L type $L_1 \cap L_2$, L_2 must be both a monotype and *upward-closed*. A type X is upward-closed if $X = \{y | y \geq x \in X - \{\bot\}\}$. In an R type $R_1 \cup R_2$, it must be the case that $R_1 \cap R_2 = 0$ in all assignments of the variables in R_1 and R_2.

The definitions of L and R types rule out certain forms of constraints for which it is not known how to perform constraint resolution. The problematic cases are an intersection on the left (i.e., $L_1 \cap L_2 \subseteq R_1$) and a union on the right (i.e., $L_1 \subseteq R_1 \cup R_2$). Thus, L types restrict intersections (on the left-hand sides of constraints) and R types restrict unions (on the right-hand sides of constraints).

A *proper* system of constraints has the form $\{L_i \subseteq R_i\}$. In [3], an algorithm is given for solving proper systems of constraints. To extend the result of [3] to include conditional types, we extend the definition of L types:

$$L ::= 0 \mid 1 \mid \alpha \mid c(L_1, \ldots, L_n) \mid R \rightarrow L \mid L_1 \cap L_2 \mid L_1 \cup L_2 \mid L_1 ? L_2$$

To show that proper systems with conditional types can be solved, it is sufficient to show how to decompose a constraint of the form $L_1 ? L_2 \subseteq R$ into smaller constraints while preserving the set of solutions. From the semantics of ?-types it follows that

$$\{L_1 ? L_2 \subseteq R\} \equiv \{L_1 \subseteq R\} \vee \{L_2 \subseteq 0\}$$

where $\equiv$ means that the two sides have the same set of solutions and $\vee$ means the union of the solutions of the two systems. The new constraints $L_1 \subseteq R$ and $L_2 \subseteq 0$ both have the correct form $L \subseteq R$. Finally, the constraints generated in a most general derivation (Definition 5.2) are proper under the extended definition.