# Soft Typing

Robert Cartwright, Mike Fagan*
Department of Computer Science
Rice University
Houston, TX 77251-1892

## Abstract

Type systems are designed to prevent the improper use of program operations. They can be classified as either *static* or *dynamic* depending on when they detect type errors. Static type systems detect potential type errors at compile-time and *prevent* program execution. Dynamic type systems detect type errors at run-time and *abort* program execution.

Static type systems have two important advantages over dynamic type systems. First, they help programmers detect a large class of program errors *before* execution. Second, they extract information that a compiler can exploit to produce more efficient code. The price paid for these advantages, however, is a loss of expressiveness, generality, and semantic simplicity.

This paper presents a generalization of static and dynamic typing—called *soft typing*—that combines the best features of both approaches. The key idea underlying soft typing is that a static type checker need not *reject* programs that contain potential type errors. Instead, the type checker can insert explicit run-time checks around "suspect" arguments of primitive operations, converting dynamically typed programs into statically type-correct form. The inserted run-time checks identify program phrases that *may* be erroneous. For soft typing to be effective, the type system must avoid inserting unnecessary run-time checks. To accomplish this objective, we have developed an extension of the **ML** type system supporting union types and recursive types that assigns types to a wider class of programs than **ML**. We have also developed an algorithm for frugally inserting run-time checks in programs that do not type check.

*The work of both authors was partially supported by NSF and DARPA.

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

## 1 Introduction

Computations in high-level programming languages are expressed in terms of operations on abstract values such as integers, matrices, sequences, trees, and functions. Most of these operations are *partial*: they are defined only for inputs that satisfy certain constraints. For example, the **head** operation on lists (**car** in LISP) is defined only for non-empty lists.

Since the programs submitted to a language translator may contain undefined applications, the language must provide some mechanism for coping with them. For this reason, most programming languages impose a type discipline on program text. This discipline can either be *static* or *dynamic*. A statically typed language prevents undefined applications by enforcing syntactic restrictions on programs that guarantee every application is defined. The language translator identifies ill-formed applications and refuses to execute programs containing them. **ML**[19] is a prominent example of a statically typed language.

In contrast, a dynamically typed language accepts all programs. Undefined applications are detected during program execution by tests embedded in the code implementing each primitive operation. If an operation's arguments are not acceptable, the operation generates a program exception, transferring control to an exception handler in the program or the operating system. Scheme[5] is a prominent example of a dynamically typed programming language.

Static typing has two important advantages over dynamic typing. They are:

**Error Detection:** The system flags all "suspect" program phrases before execution, helping the programmer detect program errors early in the programming process.

**Optimization:** The compiler can produce better code by optimizing the representation of each type and eliminating *most* run-time checks.

However, these advantages are achieved at the loss of:

**Expressiveness:** No type checker can decide whether or not an arbitrary program will generate any undefined applications. Therefore, the type checker must err on the side of safety and reject some programs that do not contain any semantic errors.

**Generality:** Some abstractions cannot be encapsulated as procedures because they do not type check. As a result, a programmer may be forced to write many different instances of the same abstraction. In Pascal, it is impossible to write a sort procedure applicable to arrays of different lengths; a separate sort procedure must be written for each array index set. In **ML**, it is impossible to write the polyadic `taut` function presented in Section 6; a separate function must be written for each arity.

**Semantic Simplicity:** The type system is a complex set of syntactic rules that a programmer must master to write correct programs. Otherwise, he will repeatedly trip over the syntactic restrictions imposed by the type checker.

In this paper, we present a generalization of static and dynamic typing—called *soft typing*—that combines the best features of both approaches. A soft type system can statically type check *all* programs written in a dynamically typed language because the type checker is permitted to insert explicit run-time checks around the arguments of "suspect" applications. In other words, the type checker transforms source programs to type-correct programs by judiciously inserting run-time checks.

Soft typing retains the advantages of static typing because the final result is a type correct program in a sublanguage that conforms to a static type discipline. In the context of soft typing, a programmer can detect errors before program execution by inspecting the program phrases where the type checker fails and inserts run-time checks. Compilers can generate efficient code for softly typed languages because the transformed program statically type checks.

The key technical obstacle to constructing a *practical* soft type system is devising a type system that satisfies two conflicting goals. The system must be

- rich enough to type "most" program components written in a dynamic typing style without inserting any run-time checks, yet
- simple enough to accommodate *automatic type assignment* (often called *type inference* or *type reconstruction*).

Existing static type systems fail to sat'sfy the first test, namely, the typing of most program components written in a dynamic style.

In this paper, we show how to construct a practical soft type system that satisfies the two preceding criteria. It is a generalization of the **ML** type system based on an encoding technique for type expressions developed by Didier Rémy[21] for typing records with inheritance. We use a variation on Rémy's encoding to represent arbitrary union types and recursive types composed from primitive type constructors. This encoding reduces the type assignment problem in our soft type system to the type assignment problem in **ML**, provided that circular unification is used instead of ordinary unification. Consequently, we can use *exactly the same type assignment algorithm* as **ML** if we subsitute circular unification for ordinary unification. Rémy's encoding also enables us to apply the **ML** type assignment algorithm on to transformed problem to determine what run-time checks to insert in programs that fail to type check.

The rest of the paper is organized as follows. Section 2 presents a simple functional programming language that serves as a basis for our study of soft typing. Section 3 identifies the appropriate design criteria for a soft type system, focusing on the necessary elements in the type language. Section 4 describes a type language that meet these criteria. Section 5 presents an inference system for deducing types for program expressions. Section 6 gives an algorithm that automatically assigns types to program expressions. Section 7 describes a method for inserting run-time checks when the type assignment algorithm fails. Finally, Section 8 discusses related work, and Section 9 presents some directions for further research.

## 2 A Simple Programming Language

For the sake of simplicity, we will confine our attention in this paper to the simple functional language **Exp** introduced by Milner[18] as the functional core of **ML**. The automatic type assignment and coercion insertion methods that we present for **Exp** can be extended to assignment and advanced control structures using the same techniques that Tofte[24], MacQueen[3], and Duba et al[10] have developed for **ML**.

We define the syntax of **Exp** as follows:

**Definition 1** [The programming language] Let $x$ range over a set of variables and $c$ range over a set of constants $K$. A program $M$ has the form:

$$M ::= x \mid c \mid \lambda x.M \mid (M\ M) \mid \texttt{let}\ x = M\ \texttt{in}\ M$$

We presume that the set of constants $K$ contains *data constructors* that build values, *selectors* that tear apart values, and **case** functions that conditionally combine operations. For example, the boolean constants **true** and **false** are 0-ary constructors, the list operation **cons** is a binary constructor, and the corresponding **head** and **tail** functions are selectors on non-empty lists. For each non-repeating sequence of constructors, there is a **case** function. If $\langle c_1, \ldots, c_n\rangle$ is a sequence of constructor names, then $\texttt{case}_{\langle c_1, \ldots, c_n\rangle}$ takes $n+1$ arguments: a value $v$, and $n$ functions $o_1, \ldots, o_n$. The $\texttt{case}_{\langle c_1,\ldots,c_n\rangle}$ function is informally defined by the following equation:

$$\texttt{case}_{\langle c_1, \ldots, c_n\rangle}(v)(o_1)\ldots(o_n) = \begin{cases} o_1(v) & v = c_1(\ldots) \\ \ldots & \\ o_n(v) & v = c_n(\ldots) \\ \textbf{wrong} & \text{otherwise} \end{cases}$$

The program text $\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z$ is syntactic sugar for the application $\texttt{case}_{\langle\texttt{true},\texttt{false}\rangle}(x)(\lambda x.y)(\lambda x.z)$. ternary **if** construct is syntactic sugar for the function $\texttt{case}_{\langle\texttt{true},\texttt{false}\rangle}$.

**Exp** is a conventional *call-by-value* functional language. All of the primitive and defined functions in **Exp** are *strict*: they diverge if any of their arguments diverge. The formal semantic definition for **Exp** is given in Appendix A. The only unusual feature of the semantics is the inclusion of a special element **wrong** in the data domain to model failed applications which can never be executed in type correct programs. The data domain also includes exception values to model the output of dynamic run-time checks.

## 3 Criteria for a Soft Type System

From the perspective of a programmer, soft typing is a mechanism for statically detecting potential type errors in dynamically typed programs. From the perspective of a compiler-writer, soft typing is a mechanism for translating dynamically typed programs to equivalent statically typed programs, reducing the problem of compiling a dynamically typed langauge to compiling a statically typed subset.

These two complementary perspectives on soft typing dictate that a practical soft type system must satisfy the following two criteria:

**Minimal Text Principle** The system should accept *unannotated* dynamically typed programs. Otherwise, the programming interface will be more cumbersome than that provided by a conventional dynamically typed programming language.

**Minimal Failure Principle** The type system should be rich enough to assign static types to "typical" program components that produce no run-time type errors. Otherwise, programmers will ignore the run-time checks inserted by the type checker because most of them are false indicators of potential program errors.

To satisfy the minimal text principle, the soft type system must perform *automatic type assignment.* Automatic type assignment deduces types for all program phrases by propagating the type information associated with primitive operations.

To satisfy the minimal failure principle, a soft type system must accommodate *parametric polymorphism*, an important form of abstraction found in dynamically typed languages. Parametric polymorphism is best explained by giving an example. Consider the well-known LISP function **reverse** that takes a list as input and reverses it. For any type $\alpha$, **reverse** maps the type $\textbf{list}(\alpha)$ to $\textbf{list}(\alpha)$. To propagate precise type information about a specific application of **reverse**, we need to capture the fact that the elements of the output list belong to the same type as the elements of the input list. Otherwise, we will not be able to type check subsequent operations on the elements of the output list. For this reason, a soft typing system must be able to deduce the fact that **reverse** has type $\forall\alpha.\textbf{list}(\alpha) \rightarrow \textbf{list}(\alpha)$.

The **ML** type system supports automatic type assignment and parametric polymorphism, but it still falls short of satisfying the minimal-failure criterion. There are two important classes of program expressions that commonly occur in dynamically typed programs that do not type-check in the ML type system.

### 3.1 Heterogeneity

The first class of troublesome program expressions is the set of expressions that do not return "uniform" results. Consider the following sample function definition.

**Example 1** Let $f$ be the function

$$\lambda x.\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else nil}$$

where **if-then-else** has type $\forall\alpha\ \textbf{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$. $f$ takes a boolean value and returns either 1 or **nil**. Clearly, no run-time error occurs if $x$ is a boolean. Nevertheless, this function fails to type check, because 1 and **nil** belong to different types.

To assign types to heterogeneous expressions, we need to introduce *union* types. In a type system with union types, the function in Example 1 has type **bool** $\rightarrow$ **int** + **nil** where **int** + **nil** denotes the *set theoretic* union of **int** and **nil**: $\{x | x \in \textbf{int} \text{ or } x \in \textbf{nil}\}$. Union types also enable us to make finer grained type distinctions because we can partition inductively defined types into unions of subtypes that distinguish how elements are constructed. The following example is a good illustration.

**Example 2** The type **list** is the union of two different constructor types: the empty list `nil` and non-empty lists $\textbf{cons}(\alpha)$. The division of the type $\textbf{list}(\alpha)$ into the union of the types **nil** and $\textbf{cons}(\alpha)$ is important because it enables us to assign more precise types to the selector functions `head` and `tail`. `head` has type $\textbf{cons}(\alpha) \rightarrow \alpha$ and `tail` has type $\textbf{cons}(\alpha) \rightarrow \textbf{list}(\alpha)$. Languages without union types typically define the type of `head` as $\textbf{list}(\alpha) \rightarrow \alpha$ and the type of `tail` as $\textbf{list}(\alpha) \rightarrow \textbf{list}(\alpha)$ permitting undefined applications such as (`head nil`) and (`tail nil`) to pass the type checker.

### 3.2 Recursivity

The other class of troublesome program expressions is the set of expressions that induce recursive type constraints. A classic example of this phenomenon is the self application function $S = \lambda x.(x\ x)$ . Since $x$ is applied as a function in the body of $S$, $x$ must have type $\alpha \rightarrow \beta$. But the self-application also forces $\alpha \rightarrow \beta$ to be subtype of the input type of $x$ which is $\alpha$. In a dynamically typed language like Scheme, the function $S$ can be written and executed. If $S$ is applied to the identity function or a constant function, it will produce a well-formed answer. Nearly all static type systems, however, reject $S$.

The simplest way to construct solutions to recursive type constraints is to include a fixed-point operator `fix` in the language of type terms. This feature is usually called *recursive* typing. Given the operator `fix`, we can assign the type $\texttt{fix}\ t.t \rightarrow \beta$ to $S$.

In practice, recursive types naturally arise in conjunction with union types. The following simple example (taken from an introductory Scheme course) is a good illustration of this phenomenon.

**Example 3** Let the function `deep` be defined by the equation

```
deep =
    λn.(if (=? n 0) 0
         (cons (deep (pred n)) nil))
```

where **0** is the 0-ary constructor denoting 0 and `pred` is the standard predecessor function.

The `deep` function takes a natural number $n$ as input and returns 0 nested inside $n$ levels of parentheses. It does not type check in **ML** because the output type is both heterogeneous and recursive. In a type system with parametric types, union types, and recursive types, `deep` has type $\textbf{nat} \rightarrow \texttt{fix}\ t.\textbf{0} + \textbf{cons}(t)$ where $\textbf{nat} = \textbf{0} + \textbf{suc}$.

## 4 A Type Language Suitable for Soft Typing

Our type language, called **Typ**, is a generalization of the **ML** type language. The definition of **Typ** is parameterized by the set of data constructors in **Exp**. We assume that all data constructors are disjoint: no value $v$ can be generated by two different constructors. For each data constructor **c** in **Exp**, we define a type constructor with the same name **c**. The set of type constructors $C$ consists of the type constructors derived from data constructors plus the special type constructor $\rightarrow$ for defining function types. *The arity of each type constructor depends on the degree of polymorphism inherent in the corresponding data constructor.* For example, the data constructor **cons** for building lists accepts arbitrary values as its first argument, but the second argument must be a list, so the **cons** type constructor has arity 1. Similarly, the data constructor **suc** takes one argument, but it must be a non-negative integer, so the type constructor **suc** has arity 0. The special type constructor $\rightarrow$ has arity 2.

Before we define the syntax of **Typ**, we need to define several subsidiary notions including *polyregular type expressions* and *tidy type expressions*.

**Definition 2** Given a set of type constructors $C$, we define the set of *type terms* $\Sigma(C, V)$ over a set of variables $V$ to be the free term algebra over $C$ and $V$. The *polyregular type expressions*[1] over $C$ and $V$ is the set of expressions inductively defined by the equation

$$T = V \mid \mathbf{c}(\ldots) \mid T + T \mid \texttt{fix}\ x.T$$

where $x$ is any type variable in $V$, **c** is any type constructor in $C$, and ... is a list of $n$ polyregular type expressions where $n$ is the arity of $c$. To avoid ambiguity, we view expressions as trees rather than strings.

The intended meaning of polyregular type expressions is the obvious one. Every closed type expression denotes a set of data objects. For each data constructor $\mathbf{c} \in C$, the corresponding type construction

[1] An explanation for the choice of this terminology and a discussion of the properties of polyregular types appears in Fagan's dissertation[11].

$c(\alpha_1, \ldots, \alpha_m)$ denotes the set of all data objects of the form $c(x_1, \ldots, x_n)$ where $(i)$ each argument $x_i$ corresponding to a *polymorphic* parameter $y_i$ of c is taken from the type denoted by the matching[2] argument $\alpha_j$ and $(ii)$ each *non-polymorphic* argument $x_i$ is taken from the type for $y_i$ specified in the definition of the data constructor c. The function type construction $\alpha \rightarrow \beta$ denotes the set of all continuous functions that map $\alpha$ into $\beta$. The $+$ operator denotes the union operation on sets and the `fix` operator denotes the fixed-point operation on type functions. A formal definition of the semantics of type expressions based on the ideal model[16] of MacQueen, Plotkin, and Sethi is given in Appendix B.

### 4.1 Tidy Type Expressions

For technical reasons, we must impose some modest restrictions on the usage of the `fix` and $+$ operators in type expressions.

**Definition 3** The set of *tidy* type expressions over $C$ and $V$ is the set of polyregular type expressions over $C$ and $V$ that satisfy the following two constraints:

1. Every subexpression of the form `fix` $v.t$ is *formally contractive*. Almost all uses of `fix` that arise in practice are formally contractive, but the formal definition is tedious. It is given in Appendix B. The expressions `fix` $x.x$ and `fix` $x.x + \mathbf{nil}$ are not formally contractive, but `fix` $x.x \rightarrow y$ and `fix` $x.\mathbf{nil} + \mathbf{cons}(x)$ are.

2. Every subexpression of the form $u + v$ is *discriminative*. A polyregular type expression of the form $u + v$ is *discriminative* iff $(i)$ neither $u$ or $v$ is a type variable, and $(ii)$ each type constructor appears *at most once* at the top level (not nested inside another type constructor) of $u + v$. The expressions $t + \mathbf{true}$, $\mathbf{cons}(x) + \mathbf{cons}(y)$, and $(\texttt{fix}\ x.\mathbf{cons}(x)) + \mathbf{cons}(y)$ are not discriminative, but $\mathbf{false} + \mathbf{true}$ and $\mathbf{nil} + \mathbf{cons}(y)$ are.

### 4.2 Type Schemes

The final step in defining our type language **Typ** is adding the notion of universal quantification. As in **ML**[8], we restrict universal quantification to the "outside" of type expressions. Tidy type expressions that may contain quantifiers are called *tidy type schemes*.

**Definition 4** [The Language **Typ**] Let $C$ be the set of type constructors for **Exp** and let $V$ be a set of type variables. A *tidy type scheme* over $C$ and $V$ is defined by the grammar:

$$\sigma ::= \tau \mid \forall t.\sigma$$

where $\tau$ denotes the set of tidy type expressions over $C$ and $V$. **Typ**, the *soft type language* for **Exp** is the set of tidy type schemes over $C$ and $V$. The **ML** *type language* for **Exp** is the set of type schemes in **Typ** that do not contain the $+$ operator (union) or the `fix` operator. Hence, the **ML** *type language* is defined by the same grammar as **Typ**, except that the symbol $\tau$ is restricted to the set of type terms.

The types of polymorphic operations are given by type schemes rather than type expressions. The formal semantics for type expressions given in Appendix B can easily be extended to type schemes by interpreting $\forall t.\sigma$ as the intersection of all ideals $\sigma$ where $t$ is an arbitrary ideal [16].

## 5 Type Inference

To assign types to programming language expressions, we use a type inference system, called **Inf**, similar to the type inference system for ML. **Inf** relies on two auxiliary relations on type expressions: $(i)$ the *generic instance* relation $\leq$ on type schemes defined by Damas and Milner[8] for the ML type system and $(ii)$ a subtyping relation $\subseteq$ on union types, which we will subsequently define.

**Definition 5** A *generic instance* of type scheme $\sigma$ is a type scheme $\sigma'$ obtained by substituting tidy type expressions for *quantified* variables of $\sigma$. For example, given the type $\forall\alpha.\mathbf{cons}(\alpha) \rightarrow \alpha$, then $\mathbf{cons}(\mathbf{int}) \rightarrow \mathbf{int}$ is a generic instance.

**Definition 6** The *subtyping* relation $\subseteq$ is defined on tidy type expressions by the inference system given in Figure 1 where $\tau, \tau', \tau_i$, and $\tau_i'$ denote tidy type expressions; $x$ and $y$ denote type variables; and $c$ denotes an $n$-ary type constructor other than $\rightarrow$. The inference system presumes that the $+$ operator is associative, commutative, and idempotent.

The subtyping inference system is decidable using the encoding defined in the next section. Given tidy type expressions $\tau$ and $\tau'$, $\tau \subseteq \tau'$ iff the supertype encoding of $\tau$ circularly unifies with the subtype encoding of $\tau'$.

Given the two auxiliary relations defined above, our type inference system **Inf** is defined as follows:

**Definition 7** [The type inference system **Inf**] A *typing judgment* is a formula $e : \sigma$ where $e$ is a program expression and $\sigma$ is a type scheme. The intended meaning of $e : \sigma$ is that $e$ has type $\sigma$. Figure 2

[2]Type argument $\alpha_j$ corresponds to the $j$th *polymorphic* parameter $y_i$ of the data constructor c. Hence, $j$ may be less than $i$.

$$
\begin{array}{ll}
\textsc{Axiom}: & A, \tau_1 \subseteq \tau_2 \vdash \tau_1 \subseteq \tau_2 \\
\textsc{Refl}: & A \vdash \tau \subseteq \tau \\
\textsc{Union}: & A \vdash \tau_1 \subseteq \tau_1 + \tau_2 \\
\textsc{Trans}: & \dfrac{A \vdash \tau_1 \subseteq \tau_2 \qquad A \vdash \tau_2 \subseteq \tau_3}{A \vdash \tau_1 \subseteq \tau_3} \\
\textsc{Con}: & \dfrac{A \vdash \tau_1 \subseteq \tau_1' \ldots A \vdash \tau_n \subseteq \tau_n'}{A \vdash c(\tau_1, \ldots, \tau_n) \subseteq c(\tau_1', \ldots, \tau_n')} \\
\textsc{Fun}: & \dfrac{A \vdash \tau_1' \subseteq \tau_1 \qquad A \vdash \tau_2 \subseteq \tau_2'}{A \vdash \tau_1 \to \tau_2 \subseteq \tau_1' \to \tau_2'} \\
\textsc{Fix1}: & A \vdash \mathtt{fix}\ x.\tau = \tau[x \leftarrow \mathtt{fix}\ x.\tau] \\
\textsc{Fix2}: & \dfrac{A, x \subseteq y \vdash \tau_1 \subseteq \tau_2}{A \vdash \mathtt{fix}\ x.\tau_1 \subseteq \mathtt{fix}\ y.\tau_2}
\end{array}
$$

Figure 1: Inference rules for subtype relation

$$
\begin{array}{lll}
\textsc{Taut}: & A \vdash x : \sigma & A(x) = \sigma \\
\textsc{Inst}: & \dfrac{A \vdash x : \sigma}{A \vdash x : \sigma'} & \sigma' \leq \sigma \\
\textsc{Gen}: & \dfrac{A \vdash e : \sigma}{A \vdash e : \forall x \sigma} & x \text{ not free in } A \\
\textsc{Abst}: & \dfrac{A \cup \{x : \tau'\} \vdash e_1 : \tau}{A \vdash \lambda x.e_1 : \tau' \to \tau} & \\
\textsc{App}: & \dfrac{A \vdash f : \tau_1 \to \tau_2 \quad A \vdash e : \tau_1}{A \vdash (f\ e) : \tau_2} & \\
\textsc{Let}: & \dfrac{A \vdash e_1 : \sigma \quad A \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau} & \\
\textsc{Sub}: & \dfrac{A \vdash e : \tau}{A \vdash e : \tau'} & \tau \subseteq \tau'
\end{array}
$$

Figure 2: The type inference rules

presents the rules in the inference system **Inf** for typing judgments. In the figure, $\sigma$ and $\sigma'$ denote tidy type schemes; $\tau, \tau', \tau_1$, and $\tau_2$ denote tidy type expressions; $e, e_1, e_2$, and $f$ denote program expressions; $x$ denotes a type variable; and $A$ denotes a set of type assumptions of the form $x : \sigma$. The intended meaning of the notation $A \vdash e : \tau$ is that $A$ implies $e : \tau$.

**Inf** consists of the conventional ML rules augmented by the Sub rule to handle subtyping. The effect of adding the Sub rule is surprisingly subtle. The most important consequence is that program expressions do not necessarily have best (*principal*) types. The following counterexample demonstrates this fact.

**Example 4** [Multiple Typings] Let $f_1 : (\alpha \to \alpha) \to \alpha \to \alpha$ and $f_2 : \mathbf{a} + \mathbf{b} \to \mathbf{a}$ be program functions where $\mathbf{a}$ and $\mathbf{b}$ are 0-ary type constructors. We can immediately deduce that $(f_1\ f_2) : \mathbf{a} \to \mathbf{a}$. Alternatively, $A \vdash (f_1\ f_2) : \mathbf{a} + \mathbf{b} \to \mathbf{a} + \mathbf{b}$. These two typings for $(f_1\ f_2)$ are incomparable, so neither is best. Furthermore, an exhaustive case analysis reveals that $\forall x(x \to x)$ is the only tidy type that has both types as supertypes, but it is not a valid typing for $(f_1\ f_2)$.

On the other hand, **Inf** retains the most important property of the ML type inference system, namely the soundness theorem and its corollaries. The soundness theorem simply asserts that every provable typing judgment is true, according to our semantics for the programming language, the type language, and type assertion language. On an informal level, the soundness theorem ensures that well-typed programs never execute undefined function applications.

## 6 Automated Type Assignment

To satisfy the minimal-text criterion for soft typing, we must produce a practical algorithm for assigning types to program expressions. We have developed an algorithm based on the type assignment algorithm from **ML** (Milner's algorithm W). Our algorithm differs from the **ML** algorithm in two respects.

- First, we encode all program types as instances of a single polymorphic type that has type parameters for every type constructor. Our encoding is based on an encoding developed by Didier Rémy[21] to reduce inheritance polymorphism on record types to parametric polymorphism.
- Second, we use circular unification[6] instead of regular unification to solve systems of type equations. Circular unification is required to infer recursive types.

Our encoding for tidy type expressions exploits the fact that the set of type constructors is *finite*. Given a finite set of type constructors, we can enumerate (up to substitution instances) all the tidy subtypes and supertypes of a given type expression. Consider the following example.

**Example 5** Assume that the only constructors in our type language are $\mathbf{a}(x), \mathbf{b}, \mathbf{c}$. Then the supertypes of the type $\mathbf{b}$ are $\{\mathbf{b}, \mathbf{a}(x)+\mathbf{b}, \mathbf{b}+\mathbf{c}, \mathbf{a}(x)+\mathbf{b}+\mathbf{c}\}$ . We can describe this set using a simple pattern by analyzing which type constructors *must* appear in a supertype, which type constructors *may* appear in a supertype, and which type constructors *must not* appear in a supertype.

The following table presents the results of this analysis:

| Status **a** | Status **b** | Status **c** |
|---|---|---|
| may | must | may |

The same form of analysis can be used to construct a pattern describing all the possible subtypes of a given type. Since **b** has no subtypes other than itself, let's consider the type $\mathbf{b}+\mathbf{c}$ instead. The subtypes of $\mathbf{b}+\mathbf{c}$ are $\{\mathbf{b},\mathbf{c},\mathbf{b}+\mathbf{c}\}$ . A "must/may/must-not" analysis of this set yields the table

| Status **a** | Status **b** | Status **c** |
|---|---|---|
| must-not | may | may |

Note that in supertype patterns, positive ("must") information plays a crucial role, while in subtype patterns, negative ("must-not") information is crucial. We need both forms of patterns because the function type constructor $\rightarrow$ inverts the form of information provided by its first argument.

We can express these patterns in the framework of an **ML** type system by using the following representation adapted from a similar translation that Rémy developed for record types with inheritance. Consider an **ML** type system with three type constructors: a highly polymorphic type constructor $\mathcal{R}$ and two type constants $+$ and $-$ signifying "must" and "must-not" respectively. $\mathcal{R}$ has a "flag" parameter for each type constructor in **Typ** and a "pattern" parameter for each type argument taken by a constructor in **Typ**. A flag parameter must be instantiated by either $+$ or $-$ A pattern parameter must be instantiated by a expression of the form $\mathcal{R}(\ldots)$.

For the type language described in the preceding example, the $\mathcal{R}$ type constructor takes four arguments: the flag for **a**, the pattern describing the type argument of **a**, the flag for **b**, and the flag for **c**. Using this notation, we can encode the patterns presented in the example as follows. The type **b** is encoded by the type term $\mathcal{R}(\mathbf{a}:-,\mathbf{1_a}:x,\mathbf{b}:+,\mathbf{c}:-)$ where $x$ is a free type variable and $\mathbf{a},\mathbf{1_a},\mathbf{b}$, and **c** are field names included as annotation. $\mathbf{1_a}$ is the field for the type parameter for **a**. The value of pattern variable $x$ is irrelevant because the corresponding constructor (**a**) is excluded from the encoded type. The supertypes of **b** are represented by the term $\mathcal{R}(\mathbf{a}:\tau_1,\mathbf{1_a}:x,\mathbf{b}:+,\mathbf{c}:\tau_2)$; every instantiation of the flag variables yields the representation of a supertype of **b**. Similarly, the type $\mathbf{b}+\mathbf{c}$ is encoded as the term $\mathcal{R}(\mathbf{a}:-,\mathbf{1_a}:x,\mathbf{b}:+,\mathbf{c}:+)$. The subtypes of $\mathbf{b}+\mathbf{c}$ are represented by the term $\mathcal{R}(\mathbf{a}:-,\mathbf{1_a}:x,\mathbf{b}:\tau_3,\mathbf{c}:\tau_4)$; every instantiation of the flag variables yields the representation of a subytype of $\mathbf{b}+\mathbf{c}$.

Any non-recursive tidy type expression $\tau$ in **Typ** can be encoded as a type term $\tau'$ over the three type constructors $\mathcal{R},+,-$, where the the set of type parameters for the type constructor $\mathcal{R}$ is tailored to the set of of type constructors in **Typ**. More importantly, the set of all supertypes of $\tau$ can be encoded as a type term $\tau^+$ over $\mathcal{R},+,-$. The encodings of the supertypes of $\tau$ are simply the terms generated by instantitating the flag variables of $\tau^+$. Similarly, the set of all subtypes of $\tau$ can be encoded as a type term $\tau^-$ over $\mathcal{R},+,-$. The encodings of the subtypes of $\tau$ are the terms generated by instantiating the type variables of $\tau^-$.

If we extend Rémy's notation to include quantification and the `fix` operator, then we can express all tidy type schemes and their corresponding sets of supertypes and subtypes using Rémy's notation.

Rémy's encoding technique has two important technical properties:

- It reduces the supertyping and subtyping relations to the instantiation of flag fields in the polymorphic type constructor $\mathcal{R}$.
- It *eliminates* the union operator $+$ from type expressions.

The combination of these two properties permit us to infer tidy types for **Exp** programs using *conventional* **ML** type inference augmented by a rule to unfold applications of the `fix` operator. For every primitive operation $f:\sigma$ in **Exp**, we map it to the Rèmy type that encodes all of the *supertypes* of $\sigma$ because every supertype of $\sigma$ is a *valid typing* for $\sigma$. By the SUB rule, $f:\sigma$ and $\sigma\subseteq\sigma'$ implies $f:\sigma'$. Consequently, any type inference that we perform using conventional **ML** type inference augmented by the `fix` rule is sound. More importantly, we can use the circular generalization of the ML type assignment algorithm to assign types to program expressions in **Exp**.

The circular generalization of the **ML** type inference system and type assignment algorithm to handle recursive types is based on work of Huet[15]. This extension of **ML** type inference is well known in the **ML** community, but it has not been added to **ML** because recursive types are not very useful in the absence of union types.

## 6.1 The Parametric Type Language

We now present the definition of the **ML** type language in which we encode tidy type expressions. Although the language only contains three constructors, the definition is tedious because the constructor $\mathcal{R}$ has so many type parameters.

**Definition 8** [Parametric Type Language] Given a set of type constructors $C$ including $\rightarrow$ and a set of type variables $V$, the set of *parametric type terms* over $C$ and $V$ is the set of type terms $\Sigma(C', V)$ where $C'$ consists of the 0-ary constructors $\{+, -\}$ and the $\mathcal{R}$ constructor of arity

$$N = \sum_{c \in C} (1 + arity(c)).$$

We assume that $C$ has the form $\{c_k | 1 \leq k \leq n\}$ where $c_1$ is the 2-ary function type constructor $\rightarrow$. The *flag position* for type constructor $c_k$ (denoted $p_k$) in the parameter list for $\mathcal{R}$ is given by the formula

$$p_k = \sum_{i=1}^{k} 1 + arity(c_i).$$

The *type parameter positions* for constructor $c_k$ are $p_k + 1, \ldots, p_k + arity(c_k)$. A term $t_i$ in $\mathcal{R}(t_1, \ldots, t_n)$ is a *flag subterm* iff $i$ is a flag position for some type constructor $c_k$. A subterm $\rho'$ of a parametric type term $\rho$ is a *flag subterm* iff it is a flag subterm of some application of $\mathcal{R}$ in $\rho$. Any subterm of $\rho$ that is not a flag subterm is called a *type subterm.* A parametric type term $\rho$ is *well-formed* iff

1. All flag subterms of $\rho$ are type variables or constants $+$ and $-$.
2. All type subterms of $\rho$ are type variables or applications of $\mathcal{R}$.
3. The type variables occurring as flag subterms in $\rho$ are disjoint from the other type variables in $\rho$. The former are called the *flag variables* of $\rho$.

It is easy to show that our encoding (which we define below) always yields well-formed parametric type terms. Similarly, ML type inference system (with circular unification) preserves well-formedness. Consequently, we will omit the adjective *well-formed* when referring to parametric type terms for the sake of brevity.

## 6.2 Definition of the Encoding Functions

The preceding discussion indicated that we can map a tidy type expression $\tau$ to corresponding parametric type terms $R_+(\tau)$ and $R_-(\tau)$, respectively denoting the supertypes and the subtypes of $\tau$. However, the definitions of the encoding functions $R_+$ and $R_-$ are not as straightforward as the preceding analysis suggests because the function type constructor $\rightarrow$ is anti-monotonic in its first argument. In the definitions of $R_+$ and $R_-$, we must treat $\rightarrow$ as a special case and make the definitions mutually recursive.

Before we define the mappings from tidy type expressions to Rémy notation, we need to introduce some subsidiary definitions.

**Definition 9** Let $||$ denote the *syntactic concatenation operator* defined by the equation:

$$\langle a_1, \ldots, a_k \rangle || \langle b_1, \ldots, b_l \rangle = \langle a_1, \ldots, a_k, b_1, \ldots, b_l \rangle$$

A tidy type expression $\tau$ of the form $c(t_1, \ldots, t_m)$ is a *component* of a tidy type expression $u + v$ (written $\tau \preceq u + v$) iff $\tau$ is identical to either $u$ or $v$, or $\tau$ is a component of either $u$ or $v$. If no term of the form $c(t_1, \ldots, t_m)$ is a component of a expression $w$, we say that $c$ *does not occur* in $w$ (written $c \not\preceq w$).

**Definition 10** [Type Encoding] We define the mappings $R_+$ and $R_-$ from tidy type expressions to parametric type terms as follows:

$$R_+(t) = \begin{cases} t & t \in V \\ \texttt{fix}\ m.R_+(t') & t = \texttt{fix}\ m.t' \\ \mathcal{R}(R_{+,1}(t) || \cdots || R_{+,n}(t)) & \text{otherwise} \end{cases}$$

$$R_-(t) = \begin{cases} t & t \in V \\ \texttt{fix}\ m.R_-(t') & t = \texttt{fix}\ m.t' \\ \mathcal{R}(R_{-,1}(t) || \cdots || R_{-,n}(t)) & \text{otherwise} \end{cases}$$

$$R_{+,1}(t) = \begin{cases} \langle +, R_-(t_1), R_+(t_2) \rangle & t_1 \rightarrow t_2 \preceq t \\ \langle \kappa_j, x'_1, x'_2 \rangle & \rightarrow \not\preceq t \end{cases}$$

$$R_{+,i}(t) = \begin{cases} \langle +, R_+(t_1), \ldots, R_+(t_m) \rangle & c_i(t_1, \ldots, t_m) \preceq t \\ \langle \kappa_j, x'_1, \ldots, x'_m \rangle & c_i \not\preceq t \end{cases}$$

$$R_{-,1}(t) = \begin{cases} \langle \kappa_j, R_+(t_1), R_-(t_2) \rangle & t_1 \rightarrow t_2 \preceq t \\ \langle -, x'_1, x'_2 \rangle & \rightarrow \not\preceq t \end{cases}$$

$$R_{-,i}(t) = \begin{cases} \langle \kappa_j, R_-(t_1), \ldots, R_-(t_m) \rangle & c_i(t_1, \ldots, t_m) \preceq t \\ \langle -, x'_1, \ldots, x'_m \rangle & c_i \not\preceq t \end{cases}$$

where each occurrence of $\kappa_j$ and $x'_i$ denotes a fresh type variable distinct from all other type variables. $\kappa_j$ signifies a flag variable while $x'_i$ signifies a pattern variable. Recall that the index of the special constructor $\rightarrow$ in $C$ is 1, which explains why the definitions of $R_{+,1}$ and $R_{-,1}$ are treated as special cases.

The encoding functions can obviously be extended to map tidy type schemes to parametric type schemes.

## 6.3 Decoding Parametric Type Terms

The inverse transformations are similar, but they contain a surprising wrinkle. In some cases, a parametric type term can denote a set of several tidy type expressions instead of just one. The unusual situation arises when the type assignment algorithm equates two flag variables.

**Example 6** Assume that the type system has four type constructors: $\rightarrow$ and three 0-ary type constructors $\mathbf{a}, \mathbf{b}, \mathbf{c}$. Their flags appear in argument positions 1, 4, 5, and 6, respectively, of the $\mathcal{R}(\ldots)$ constructor. The type arguments for $\rightarrow$ appear in positions 2 and 3. Let

$$\texttt{twice} = \lambda f \lambda x.(f\ (f\ x))$$

The function **twice** has type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. Now suppose $F$ is a function of type $a + b \rightarrow a$. Then the parametric type for $F$ is

$$\mathcal{R}(+, \mathcal{R}(-, x_1, y_1, \delta_1, \gamma_1, -), \\ \mathcal{R}(\delta_2, x_2, y_2, +, \gamma_2, \theta_2), \delta, \gamma, \theta)$$

implying that ($\texttt{twice}\ F$) has type

$$\tau = \mathcal{R}(\rightarrow: +, \mathcal{R}(-, x_1, y_1, +, \gamma_1, -), \\ \mathcal{R}(-, x_1, y_1, +, \gamma_1, -), -, -, -)$$

It is easy to show that this type is not the image of a tidy type expression under $R_+$. Both

$$\tau_1 = \mathcal{R}(+, \mathcal{R}(-, x_1, y_1, +, +, -), \\ \mathcal{R}(-, x_1, y_1, +, +, -), -, -, -)$$

$$\tau_2 = \mathcal{R}(+, \mathcal{R}(-, x_1, y_1, +, -, -), \\ \mathcal{R}(-, x_1, y_1, +, -, -), -, -, -)$$

are flag instantiations of $\tau$, implying that $\tau_1$ and $\tau_2$ denote supertypes of $\tau$. But $\tau_1$ and $\tau_2$ encode the types $\mathbf{a} \rightarrow \mathbf{a}$ and $\mathbf{a} + b \rightarrow \mathbf{a} + b$, respectively. From Example 4, we know that these two types are not supertypes of any tidy type except $\forall x(x \rightarrow x)$, which is not a valid typing for ($\texttt{twice}\ F$).

In the general case, the parametric type for a program phrase represents a *non-empty set* of tidy types that could be inferred in **Inf**. Fortunately, it is easy to determine exactly how many tidy types a parametric type represents and what they are. The decoding process translates a parametric type scheme $\rho$ into a unique tidy type scheme $\tau$ iff there is a smallest type in the set of supertypes denoted by $\rho$. Each occurrence of a flag variable contributes positively or negatively to the denoted type. The decoding process yields a single tidy type iff all the occurrences of each flag variable are either positive or negative.

**Definition 11** An occurrence of a flag variable in a parametric type term $\rho$ is *positive* iff it appears within the left hand argument of of an *even* number of $\rightarrow$ constructors. Otherwise, the occurrence is *negative*. A *flag substitution* for $\rho$ is a substitution that binds all the flag variables of $\rho$ to $+$ or $-$ and nothing else. A parametric type term $\rho$ is *univalent* iff there is a flag substitution (substitution for flag variables) that binds every positive occurrence of a flag variable to $-$ and every negative occurrence to $+$. This substitution is called the *minimizing* flag substitution. The result of applying the *minimizing* flag substitution to a univalent type term $\rho$ is called the *minimum instance* of $\rho$. parametric type term produced by applying A flag variable is a *splitting* variable in $\rho$ iff it occurs both positively and negatively. If $\rho$ is not univalent, then its *valence* is $2^k$ where $k$ is the number of distinct splitting variables in $\rho$. A *splitting substitution* for a parametric type term $\rho$ is a flag substitution that binds all variables that occur only positively to $-$, all variables that occur only negatively to $+$, and the splitting variables to either $+$ or $-$. A parametric type term is *ground* iff it contains no flag variables (type variables may still be free).

**Remark** The minimizing substitution for a univalent term is a degenerate form of splitting substitution. Hence, the number of distinct splitting substitutions for a parametric type term $\rho$ equals the valence of $\rho$.

Ground parametric type terms are easy to decode because they are the images of supertype encodings composed with minimizing substitutions. Let **min** be the function that maps univalent parametric type terms to their minimum instances. Then the decoding function $R^{-1}$ is the inverse of $\mathbf{min} \circ R_+$.

**Definition 12** [Decoding Function] The function $R^{-1}$, which maps parametric ground terms to tidy type expressions, is defined by the following equations:

$$R^{-1}(\rho) = \begin{cases} t & t \in V \\ \texttt{fix}\ x.R^{-1}(M) & \rho = \texttt{fix}\ x.M \\ \sum_i R_i^{-1}(\rho) & \text{otherwise} \end{cases}$$

$$R_i^{-1}(\rho) = \begin{cases} t & t \in V \\ \texttt{fix}\ x.R^{-1}(M) & \rho = \texttt{fix}\ x.M \\ \mathrm{c}_i(R^{-1}(t_1), \ldots, R^{-1}(t_n)) & \rho = \mathcal{R}(s_1, \ldots, s_k), \\ & s_i = \langle +, t_1, \ldots, t_n \rangle \\ \emptyset & \text{otherwise} \end{cases}$$

Given the preceding definitions, it is easy to define the decoding translation of any parametric type term $\rho$. The translation consists of two steps. First, generate the set $S$ of all of the splitting flag substitutions for $\rho$. Second, for each substitution in $s \in S$, compute $R^{-1}(s\rho)$, yielding the set of tidy type expressions represented by $\rho$.

## 6.4 Type Assignment Algorithm

To recapitulate the preceding discussion, our type assignment algorithm consists of three steps.

1. Encode the set of supertypes for each primitive operation as parametric type terms (Rémy notation).
2. Perform ordinary type assignment using the circular generalization of algorithm W.
3. Translate the parametric type expressions assigned to program phrases back into (sets of) tidy type expressions.

The algorithm is reasonably efficient because it is simply the circular generalization of Algorithm W from **ML** applied to a transformed collection of type constructors and typings for the primitive operations. Circular unification like ordinary unification can be performed in linear time. Of course, the worst case running time of Algorithm W is exponential in the number of nested let declarations, but this behavior has never been observed in practice—presumably because the type expressions produced in these pathological cases are enormous and hence incomprehensible.

## 6.5 Some examples

Consider the function

$$\texttt{mixed} = \lambda\ x.\texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else nil}$$

defined in Example 1. Our algorithm assigns the type **true** + **false** → **suc** + **nil** to `mixed`.

A more interesting example is the function `taut` which determines whether an arbitrary Boolean function (of any arity!) is a tautology (`true` for all inputs).

**Example 7** [Tautology example]

```
taut = λB.case B of
      true :  true
      false:  false
      fun : ((and (taut (B true)))
                (taut (B false)))
```

For the `taut` function, our algorithm produces the typing `taut` : $\beta$ → (**true** + **false**) where $\beta$ = `fix` $t$.(**true** + **false** + ((**true** + **false**) → $t$)).

## 7 Inserting run-time checks

As we explained in the introduction, *no* sound type checker can pass all "good" programs. The polyregular type checker described in section 6 succeeds in assigning static types to a large class programs. Nevertheless, some "good" programs will not pass our type checker, as the following example demonstrates.

**Example 8** ] The function

$$N_1 = \lambda\ f.\ \texttt{if}\ f(\texttt{true})\ \texttt{then}\ f(5) + f(7)$$
$$\texttt{else}\ \mathrm{f}(7)$$

fails to type check because the parameter $f$ has to satisfy conflicting constraints. The `if` test forces $f$ : `true` → `true` + `false`. Similarly, the `true` arm of the `if` requires $f$ : `suc` → `z` + `suc`. But there is no unifier for these two typings of $f$, preventing the type checker from assigning a type to $N_1$. The function $N_1$ is not badly defined, however, because $N_1(\lambda\ x.x)$ never goes wrong.

Sometimes our type checker fails to account for all the possible uses of a function. Consider the following modification of the previous example.

**Example 9** [An anomaly]

$$N_2 = \lambda\ f.\ \texttt{if}\ f(\texttt{true})\ \texttt{then}\ f(5)\ \texttt{else}\ f(7)$$

In this case, our type assignment algorithm yields the typing $N_2$ : (**true** + **suc** → **true** + **false**) → **true** + **false**. However, the application $N_2(\lambda\ x.x)$ is well-defined, but does not type check.

In both of the preceding examples, the program is meaningful, but the type analysis is not sufficiently powerful to assign an appropriate type. The static type checker described in Section 6 will reject these programs, in spite of their semantic content. A soft type system, however, *cannot* reject programs. It must insert explicit run-time checks instead. Hence, we must produce an algorithm to transform arbitrary programs to equivalent programs that type check.

To support the automatic insertion of explicit run-time checks, we force the programming language **Exp** to include an *exceptional value* `fault` and a collection of functional constants called *narrowers*. The narrowers perform run-time checks and the exceptional value propagates the fact that a run-time check failed. To define these notions more precisely, we need to introduce the concepts of *primitive* and *simple* types. Each type constructor determines a *primitive* type: it consists of all values that belong to some instance of the type. We denote the primitive type corresponding to a constructor $c$ by the name of the constructor $c$. Since the type constructors are disjoint, every value (other than errors) belongs to *exactly one* primitive type. For values that are not functions, the primitive type of the value is the outermost constructor in the representation of the value. For a function, the primitive type is simply →. A *simple* type is simply a union of primitive types. Using notation analogous to type terms, we denote the simple type consisting of the union of primitive types $t_1, \ldots, t_n$ by the expression $t_1 + \ldots + t_n$.

The exceptional value is denoted by `fault`. Similarly, there is a narrower $\Downarrow^S_T$ for every pair of simple types such that $S \subseteq T$. It is defined by the equation

$$\Downarrow^S_T(v) = \begin{cases} v & v \in T \\ \texttt{fault} & v \in S - T \\ \texttt{wrong} & \text{otherwise} \end{cases}$$

The type associated with the narrower $\Downarrow^S_T$ is $s_1 + \ldots \rightarrow t_1 + \ldots$ where $S = \{s_1, \ldots\}$ and $T = \{t_1, \ldots\}$.

The type insertion algorithm consists of the following three steps:

1. Encode the set of supertypes for each primitive operation in parametric form using $R_+$ and convert all "−" flags in the encoding to fresh type variables—eliminating all negative information.

2. Perform type assignment using the circular generalization of algorithm W, constructing a "positive" type (in parametric form) for every program expression.

3. For each occurrence of a primitive operation, unify the "positive" type with the original type (encoded in parametric form). If unification fails, insert the narrower required for unification.

The invocation of Algorithm W in the type insertion algorithm always succeeds in assigning a type to every program expression because the only way that Algorithm W can fail is to attempt to unify a "+" flag and a "−" flag.[3] Since no "−" flags can appear in any of the type terms manipulated by the algorithm, every unification step must succeed. However, the type assignment is based on the assumption that every primitive operation explicitly checks the type-correctness of any argument that it evaluates. If the check fails, the primitive operation must return the special value `fault` as its answer. All of the primitive operations are strict with respect to this special value.[4] The final unification step determines which run-time checks are unnecessary. If the input program is statically type correct, no run-time checks are inserted.

Note that the narrower inserter algorithm is essentially the type assignment algorithm partitioned into two phases: the propagation of positive information followed by the propagation of negative information. In the second phase, narrowers are inserted where the negative information in the input side of primitive functions clashes with the positive information inferred for their arguments.

[3]Algorithm W never attempts to unify a flag variable with a pattern variable or a `fix` operation because all parametric type terms are *well-formed*.

[4]We are using the term strict in a slightly different sense than it usually is in the literature. Primitive operations must return the value `fault` if they evaluate an argument and that evaluation returns `fault` as the answer.

### 7.1 Examples

To illustrate the insertion process, we analyze how it handles the two troublesome examples that we presented at the beginning of this section. In both of these examples, let $\texttt{c}_1$ and $\texttt{c}_2$ denote the narrowers $\Downarrow^{\mathbf{z+suc+true+false}}_{\mathbf{z+suc}}$ and $\Downarrow^{\mathbf{z+suc+true+false}}_{\mathbf{true+false}}$, respectively.

The insertion algorithm changes the function $N_1$ in example 8 to

$$\begin{aligned} N_1' = \lambda\ f.\ &\texttt{if}\ \texttt{c}_2(f(\texttt{true})) \\ &\texttt{then}\ \texttt{c}_1(f(5)) + \texttt{c}_1(f(7)) \\ &\texttt{else}\ f(7) \end{aligned}$$

The type for $N_1'$ is now $(\mathbf{z} + \mathbf{suc} + \mathbf{true} + \mathbf{false} \rightarrow \mathbf{z} + \mathbf{suc} + \mathbf{true} + \mathbf{false}) \rightarrow \mathbf{z} + \mathbf{suc}$ and the application $N_1'(\lambda\ x.x)$ is now type correct.

Similarly, in example 9, assume that a program contains definition of $N_2$ and the application $(N_2\ (\lambda x.x))$. Then our insertion algorithm will modify the definition of $N_2$ to produce

$$N_2' = \lambda\ f.\ \texttt{if}\ \texttt{c}_2(f(\texttt{true}))\ \texttt{then}\ f(5)\ \texttt{else}\ f(7)$$

Now, the type system infers the type for $N_2'$ as:

$$\begin{aligned} &(\mathbf{z} + \mathbf{suc} + \mathbf{true} + \mathbf{false} \rightarrow \\ &\ \ \mathbf{z} + \mathbf{suc} + \mathbf{true} + \mathbf{false}) \rightarrow \\ &\mathbf{z} + \mathbf{suc} + \mathbf{true} + \mathbf{false} \end{aligned}$$

and $N_2'(\lambda\ x.x)$ now type checks.

## 8 Related Work

The earliest work on combining static and dynamic type checking was an investigation of the type "dynamic" conducted by Abadi et al[1]. They added dynamic data values to a conventional statically typed data domain and provided facilities for converting data values to dynamic values by performing explicit "tagging" operations. This system permits statically typed programs to manipulate dynamic forms of data. But it does not meet the criteria for soft typing because the programmer must annotate his program with explicit tagging and stripping operations to create "dynamic" values and to map them back to conventional "static" values. In addition, programs are still rejected; narrowers are not inserted by the type checker.

More recently, Thatte[23] has developed the notion of "quasi-static typing" that augments a static type system with a universal type $\Omega$ that contains tagged copies of all the values in the program data domain. Thatte's system resembles soft typing because it ensures that all programs can be executed by inserting narrowers when necessary to convert values of type $\Omega$

to corresponding values belonging to a specific type. However, it does not meet the other criteria required for soft typing for two reasons. First, it requires explicit declarations of the argument types for functions. Second, it cannot type check many dynamically typed programs (without inserting narrowers) because it does not support parametric polymorphism, recursive types, or more than one level of subtyping.

Gomard[14] has modified the **ML** type system to include a type *undefined* and dynamically typed variants of all the primitive operations. These variants accept arguments of *undefined* type. His system can perform a weak form of soft typing by replacing primitive operations by their dynamically typed variants. But the types assigned by the system are not very precise because his system does not accommodate either parametric polymorphism or union types. As a result, his system cannot statically type check many dynamically typed programs without modifying them.

Researchers in the area of optimizing compilers have developed static type systems for dynamically typed languages to extract information for the purposes of code optimization. A good example is the system developed by Aiken and Murphy[2] for **FL**. These type systems, however, have not been designed to be used or understood by programmers. Consequently, they lack the uniformity and generality required for soft typing.

Finally, many researchers in the area of static type systems have developed extensions to the **ML** type system that can type a larger fraction of "good" programs. Both Mitchell[20] and Fuh and Mishra[12, 13] have studied the problem of type inference in the presence of subtyping. They augment type expressions by explicit constraints, which permits them to infer more precise types in some cases than we do. On the other hand, their type descriptions are frequently verbose and difficult to comprehend. In addition, neither of of these systems accommodate parametric polymorphism. Wand[25] augments the **ML** by adding record operations that support a limited form of inheritance polymorphism. Curtis[7] has proposed adding constrained quantification to the **ML** type system. The idea looks promising, but it is an open question whether a suitable type assignment algorithm exists. Finally, Rémy[21] has developed a clever reduction of inheritance polymorphism to **ML** parametric polymorphism that supports parametric polymorphism in the original type language. We have heavily relied on this reduction in the construction of our soft typing system.

## 9 Directions for Further Research

In this paper, we introduced a new approach to program typing called *soft typing* that combines the best features of *static* and *dynamic* typing. To demonstrate that soft typing is feasible, we presented a soft typing system incorporating union types, recursive types, and parametric polymorphism. For this type system, we showed that there are efficient algorithms for performing automatic type assignment and automatic coercion insertion. The algorithms are both variants of the type assignment algorithm for **ML**.

We are currently engaged in a project to design a dynamic generalization of **ML** and implement the generalized language by using a soft type checker to translate it to Standard **ML**. Since Standard **ML** does not infer either union types or recursive types, the soft type checker must perform some supplementary translation. In particular, the type checker must:

- generate type definitions creating disjoint unions and tagged recursive types corresponding to all of the union types and recursive types inferred for the program, and
- insert explicit injection and projection operations to add and remove tags for disjoint unions and tagged recursive types.

We are also interested in developing new soft type systems that are richer than the one presented in this paper. In particular, we want to explore the possibility of adding some form of type intersection to the tidy polyregular types. Through this process, we may be able to construct a type system than subsumes the tidy polyregular types but possesses the principal typing property. In addition, the new type system may have simpler translations to and from Rémy notation. As we observed in the paper, the Rémy translation occasionally yields multiple polyregular types for a given program expression. In a system with intersection types, these multiple types could be expressed as a single intersection type.

## References

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. In *Proceedings of the Sixteenth POPL Symposium*, 1989.

[2] Alexander Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *Proceedings of the 18th Annual Symposium on Principles of Programming Languages*, 1991. To appear.

[3] Andrew W. Appel and David MacQueen. *Standard ML of New Jersey Reference Manual.* (in preparation), 1990.

[4] Robert Cartwright. A constructive alternative to axiomatic data type definitions. In *Proceedings of 1980 LISP Conference*, 1980.

[5] William Clinger and Jonathan Rees. *Revised$^{3.99}$ Report on the Algorithmic Language Scheme*, August 1990.

[6] Alain Colmerauer. Prolog and infinite trees. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1982.

[7] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox PARC, 1990.

[8] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982.

[9] Luis Manuel Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[10] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.

[11] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages.* PhD thesis, Rice University, 1990.

[12] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *Conference Record of the European Symposium on Programming*, 1988.

[13] You-Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *TAPSOFT*, 1989.

[14] Carsten K. Gomard. Partial type inference for untyped functional programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990.

[15] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris, 7 1976.

[16] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1983.

[17] D. B. MacQueen and Ravi Sethi. A semantic model of types for applicative languages. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, 1982.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[19] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[20] John C. Mitchell. Coercion and type inference. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1983.

[21] Dider Rémy. Typechecking records and variants in a natural extension of ml. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[22] David A Schmidt. *Denotational Semantics.* Allyn and Bacon,Inc, 1986.

[23] Sattish Thatte. Quasi-static typing. In *Proceedings of the Seventeenth POPL Symposium*, 1990.

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

[25] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the Second Symposium on Logic in Computer Science*, 1987.

## A Programming language semantics

We divide the specification of the syntax of **Exp** into two parts. The first part is the syntax of the functional language Milner introduced to analyze parametric polymorphism[18]. It is presented in definition 1 in Section 2 of the paper. That syntax is parameterized by a set of constants $K$. The second part of our specification identifies and defines these constants.

The set of constants $K$ reflects the different kinds of data included in the programming language. In **Exp**, every (non-functional) data value is built from data constructors. Data constructors may be partial: they can impose modest restrictions on the form of arguments that they accept (a simple form of dynamic typing). In addition to data constructors, $K$ contains a separate set of *selectors* for each constructor, one for each argument position of the constructor. The specification of the constructors and selectors can be succinctly expressed as a data tableau[4]. The following example illustrates this idea.

**Example 10** [A Data Tableau] Consider a programming language with natural numbers and lists as data values. We can define the construction of the natural numbers from **0** and **suc** (successor) in the usual fashion. Lists are either empty (nil) or constructions built with cons. The data tableau describing this data language is:

```
constructor  0;
constructor  suc(pred:0 + suc);
constructor  nil;
constructor
  cons(head:0 + suc + cons + nil+ →,
      tail:nil + cons)
```

The tableau designates **pred** as the selector for **suc** constructions, and **head** and **tail** as the selectors for **cons** constructions. It also indicates that the argument in a **suc** construction must be either the constant **0** or a **suc** construction. The first argument of a **cons** construction may be anything. The pseudo-constructor $\rightarrow$ designates the set of functions so that functions may appear within constructed values.

To manipulate the data values defined by a tableau, we need to include **case** functions in $K$. For every non-empty, non-repeating sequence $s$ of constructor names (including $\rightarrow$), there is an $n$-ary **case** function where $n$ is $1 + \mathit{length}(s)$. An informal description of the semantics of these functions is given in Section 2.

We define the denotational semantics for **Exp** along standard lines as presented in Schmidt[22]. The data domain $D$ for **Exp** depends on the constants specified in the data tableau. We define $D$ as follows:

**Definition 13** [The data domain $D$] Let $C = \{\mathtt{c}_i | 2 \leq i \leq n\}$ be the set data constructors (excluding $\rightarrow$) defined in the tableau and let $\mathit{Triv} = \{*, \perp\}$ be a one point domain. Let $D^0$, $\mathbf{F}$, and $\mathbf{W}$ abbreviate $\mathit{Triv}$; let $D^k$ abbreviate $D \otimes D^{(k-1)}$ where $\otimes$ is the domain smash product; and let $D^{|\mathtt{c}_i|}$ abbreviate $D^{\mathit{arity}(\mathtt{c}_i)}$. The data domain $D$ is the least solution to the equation

$$D = D \rightarrow D \oplus \ldots D^{|c_i|} \ldots \oplus \mathbf{F} \oplus \mathbf{W}$$

where $\oplus$ denotes the domain smash sum. The domains separated by $\oplus$ operators on the right hand side of the equation are called *summands* of $D$.

For each constructor (including $\rightarrow$), there is a corresponding summand in definition 13. We define

$$\begin{aligned}
[\![x]\!]\eta &= \eta(x) \\
[\![\lambda x.e]\!]\eta &= \mathit{In}_{D\rightarrow D}(\lambda d.[\![e]\!][x = d]) \\
[\![(e_1\ e_2)]\!]\eta &= \begin{cases} \mathtt{wrong} & d_1 = \mathtt{wrong} \\ \mathtt{fault} & d_1 = \mathtt{fault} \\ \mathtt{wrong} & d_2 = \mathtt{wrong} \\ \mathtt{fault} & d_2 = \mathtt{fault} \\ f_1(d_2) & \text{otherwise} \end{cases} \\
&\quad \text{where } d_1 = [\![e_1]\!]\eta), \\
&\quad d_2 = [\![e_2]\!]\eta \\
&\quad f_1 = \mathit{Out}_{D\rightarrow D}(d_1) \\
[\![\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2]\!]\eta &= [\![e_2]\!]\eta[x = d_1], \\
&\quad \text{where } d_1 = [\![e_1]\!]\eta
\end{aligned}$$

Figure 3: Semantic function for Exp

fault $= \mathit{In}_{\mathbf{F}}(*)$ and wrong $= \mathit{In}_{\mathbf{W}}(*)$. The wrong value represents undetected run-time errors, and the fault value represents safely detected run-time exceptions.

In the denotational definition for **Exp** we use the following notation. For each summand $A$, the symbol $\mathit{In}_A : A \rightarrow D$ denotes the standard injection function mapping $A$ into $D$. Similary, the symbol $\mathit{Out}_A : D \rightarrow A$ denotes the standard projection function mapping $D$ onto $A$. In addition, the conditions appearing in a definition of the form

$$f(x) = \begin{cases} \text{val}_1 & \text{condition}_1 \\ & \vdots \\ \text{val}_k & \text{condition}_k \end{cases}$$

are tested in sequential order, implying no condition can be satisfied unless the conditions preceding it fail.

**Definition 14** [Denotational Semantics of **Exp**] Let Env be the domain of *environments* Id $\rightarrow D$. The semantic function $[\![\cdot]\!] : \mathbf{Exp} \rightarrow \text{Env} \rightarrow D$ is defined in figure 3.

Before we define the semantics for constants, we need to define some auxiliary functions. Let $R = \{\mathtt{r}_1, \ldots, \mathtt{r}_n\}$ be a set of constructors. The auxiliary function $\mathcal{C}_R$ verifies that its argument belongs to one of the summands specified by $R$. It is defined by the equation:

$$\mathcal{C}_R(d) = \begin{cases} d & d = \mathit{In}_{D_{\mathtt{r}_1}}(d') \\ \vdots & \\ d & d = \mathit{In}_{D_{\mathtt{r}_n}}(d') \\ \mathtt{wrong} & \text{otherwise} \end{cases}$$

$$\begin{array}{lcll} [\![\mathtt{c}]\!] & = & In_{D_{\mathtt{C}}}(*) & n = 0 \\ [\![\mathtt{c}]\!] & = & In_{D\to D}(K) & n \neq 0 \\ [\![\mathtt{case}_{\langle \mathtt{c}_1,\ldots,\mathtt{c}_k\rangle}]\!] & = & In_{D\to D}(C) & \\ [\![\Downarrow^S_T]\!] & = & In_{D\to D}(N) & \end{array}$$

where

$$\begin{array}{lcl} K & = & \lambda d_1 \ldots d_n . \\ & & \quad In_{D_{\mathtt{C}}}(\mathcal{C}_{R_i}(\langle d_1, \ldots, d_n\rangle)) \\ & & \lambda d f_1 \ldots f_k . \\ C & = & \begin{cases} f_1(d) & d = In_{c_1}(d') \\ & \vdots \\ f_k(d) & d = In_{c_k}(d') \\ \mathtt{wrong} & \text{otherwise} \end{cases} \\ & & \lambda d . \\ N & = & \begin{cases} d & \mathcal{C}_T(d) \neq \mathtt{wrong} \\ \mathtt{fault} & \mathcal{C}_S(d) \neq \mathtt{wrong} \\ \mathtt{wrong} & \text{otherwise} \end{cases} \end{array}$$

Figure 4: Semantics for primitive operations

where $D_{\mathbf{r}_i}$ is the summand of $D$ associated with constructor $\mathbf{r}_i$.

For sets of constructors $R_i, 1 \leq i \leq k$, we define:

$$\mathcal{C}_{R_1,\ldots,R_k}(\langle d_1, \ldots, d_k\rangle) = \\ \begin{cases} \langle d_1, \ldots, d_k\rangle & \text{for all } i, 1 \leq i \leq k \\ & \mathcal{C}_{R_i}(d_i) \neq \mathtt{wrong} \\ \mathtt{wrong} & \text{otherwise} \end{cases}$$

**Definition 15** [Semantics of Constants] The semantics for constants is given in Figure 4 where $\mathtt{c}$ is an arbitrary constructor of arity $n$, $D_{\mathtt{C}}$ is the associated summand in $D$, and $R_i$ indicates the set of restrictions on the $i$-th argument of $\mathtt{c}$ declared in the data tableau. Since the meaning of constants does not depend on the environment, the environment argument is left implicit.

## B Type Semantics

Our semantics for types is based on the ideal model for recursive parametric types developed by MacQueen, Plotkin and Sethi[16]. Ideals are downward-closed, directed-closed subsets of $D$. We denote domain of ideals over $D$ by the symbol $\mathcal{I}$.

MacQueen, Plotkin and Sethi define a metric on the ideals and use that metric to define a fixed point operation on *contractive* type functions mapping $\mathcal{I}$ into $\mathcal{I}$. They show that the abstraction of a type expression

$$\begin{array}{rcl} \mathcal{T}[\![\mathtt{c}]\!]\nu & = & D_{\mathtt{C}} \cup \mathtt{fault} \\ \mathcal{T}[\![\mathtt{c}(t_1, \ldots, t_n)]\!]\nu & = & In_{D_{\mathtt{C}}}(\langle \mathcal{T}[\![t_1]\!]\nu, \ldots, \mathcal{T}[\![t_n]\!]\nu\rangle) \\ & & \cup\, \mathtt{fault} \\ \mathcal{T}[\![t_1 \to t_2]\!]\nu & = & \mathcal{T}[\![t_1]\!]\nu \Longrightarrow \mathcal{T}[\![t_2]\!]\nu \cup \mathtt{fault} \\ \mathcal{T}[\![t_1 + t_2]\!]\nu & = & \mathcal{T}[\![t_1]\!]\nu \cup \mathcal{T}[\![t_2]\!]\nu \\ \mathcal{T}[\![\mathtt{fix}\ \alpha . t]\!]\nu & = & \mu\ i . f(i) \cup \mathtt{fault} \\ & & \text{where } f(i) = \mathcal{T}[\![t]\!]\nu[\alpha = i] \end{array}$$

Figure 5: Type Semantics

that is formally contractive in the type variable $x$ with respect $x$ denotes a *contractive* function on $\mathcal{I}$.

**Definition 16** [Formal Contractiveness ] A type expression $E$ is formally contractive in $x$ iff either:

1. $E = \mathtt{c}$ for $\mathtt{c}$ a 0-ary type constructor.
2. $E = x'$ for some variable $x' \neq x$.
3. $E = \mathtt{c}(E_1, \ldots, E_n)$ for $\mathtt{c}_n$ an $n$-ary type constructor (including $\to$).
4. $E = E_1 + E_2$ where $E_1$ and $E_2$ are type expressions that are formally contractive in $x$
5. $E = \mathtt{fix}\ x'.E_1$ where $E_1$ is formally contractive in $x$ and $x'$.

**Definition 17** [Type Semantics] The semantics of the type language **Typ** is given by the meaning function $\mathcal{T}[\![]\!]$ defined in Figure 5. In the definition, $\nu : V \to \mathcal{I}$ denotes a valuation for free type variables; $In_S$ denotes the injection function for summand $S$ mapped over ideals of $S$ (interpreted as sets); $\mu$ stands for the fixed point operator on contractive functions mapping $\mathcal{I}$ into $\mathcal{I}$: $\mathtt{c}$ denotes any type constructor other than $\to$: and $\Longrightarrow$: $\mathcal{I} \to \mathcal{I} \to \mathcal{I}$ denotes the ideal function space constructor defined by the equation:

$$A \Longrightarrow B = \{f \in D \to D \mid f(A) \subseteq B\}.$$

The type semantics constructs ideals that exclude $\mathtt{wrong}$ so that undefined applications cannot type check. On the other hand, all of the ideals include the element $\mathtt{fault}$, so that narrowers have a legitimate static type.