Soft Typing

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

Summary

Type systems are designed to prevent the improper use of program operations. They can be classified as either *static* or *dynamic*. Static type systems detect "ill-typed" programs at compile-time and prevent their execution. Dynamic type systems detect "ill-typed" programs at run-time.

Static type systems have two important advantages over dynamic type systems. First, they provide important feedback to the programmer by detecting 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, modularity, and semantic simplicity.

This paper presents a *soft* type systems that retains the expressiveness of dynamic typing, but offers the early error detection and improved optimization capabilities of static typing. The key idea underlying soft typing is that a type checker need not *reject* programs containing "ill-typed" phrases. Instead, the type checker can insert explicit run-time checks, transforming "ill-typed" programs into type-correct ones.

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 of a certain form. For example, the `head` operation (`car` in LISP) is defined only for non-empty sequences.

Since programmers can write programs with undefined applications, a programming language must assign some form of meaning to undefined applications. To address this problem, most programming languages adopt a type discipline. This discipline can either be *static* or *dynamic*. A *statically typed language* prevents undefined applications by imposing syntactic restrictions on programs that *guarantee* the definedness of every application. The implementation refuses to execute programs that do not meet these restrictions. ML[17] is a prominent example of a contemporary 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 program 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:

Error Detection: The system flags all "suspect" program phrases before execution.

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

However, these advantages are achieved at the cost of a loss of expressiveness, modularity, and semantic simplicity:

Expressiveness: no type checker can always decide whether or not a 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. As a result, they reject many programs that lie beyond the limited deductive power of the type-checking system.

Modularity: some procedures in program libraries and some procedures exported from modules will be assigned more restrictive types than their meaning requires, restricting the contexts in which they can be used.

Semantic Simplicity: the type system (a complex set of syntactic rules) is an essential part of the language definition that a programmer must understand to write programs. Otherwise, he will repeatedly trip over the syntactic restrictions imposed by the type system.

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 each run-time check inserted by the type checker. Compilers can generate efficient code for softly typed languages because the transformed program is type correct.

The key technical obstacle to constructing a *practical* soft type system is devising a type system that is

- 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 satisfy the first test, namely, the typing of most program components written in a dynamic style *without inserting run-time checks.*

In this paper, we show how to construct a practical soft type system that subsumes the ML type system. Our work adapts an encoding technique developed by Rémy. This technique enables our soft type system to rely on essentially the same type assignment algorithm as ML. The only difference is that our algorithm uses circular unification instead of ordinary unification. Rémy's encoding also enables us to use the ML type assignment algorithm on a transformed problem to determine what run-time checks to insert.

The rest of the paper is organized as follows. Section 2 presents the technical preliminaries. 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 and 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 summarizes our principal results.

2 Exp: a Simple Programming Language

We are interested in developing a soft type system suitable for higher order imperative languages like Scheme and Standard ML. For the sake of simplicity, we will confine our attention in this paper to the simple functional language **Exp** introduced by Milner[16] 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[21], MacQueen[4], and Duba et al[9] have developed for ML.

The syntax of **Exp** is given by the following grammar:

Definition 1 *(The programming language)* Let x range over a set of variables and c range over a set of constants K

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

The set K of constants contains *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 `head` and `tail` functions are selectors on non-empty lists (constructed using the `cons` operation. 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 $\mathtt{case}_{\langle c_1,\ldots,c_n\rangle}$ takes $n+1$ arguments: a value v, and n functions $o_1, \ldots, o_n$. The $\mathtt{case}_{\langle c_1,\ldots,c_n\rangle}$ is informally defined by the following equation:

$$\mathtt{case}_{\langle c_1,\ldots,c_n\rangle}(v)(o_1)\ldots(o_n) = \begin{cases} o_1(v) & v \text{ is a } c_1 \text{ construction} \\ \ldots & \\ o_n(v) & v \text{ is a } c_n \text{ construction} \\ \mathtt{fatal\text{-}error} & \text{otherwise} \end{cases}$$

The standard ternary `if` operation is synonymous with the function $\mathtt{case}_{\mathsf{true},\mathsf{false}}$.

Exp is a conventional *call-by-value* functional language. With the exception of the primitive `case` functions, all functions diverge if their arguments diverge. The formal semantic definition for **Exp** is given in an Appendix in the full paper. The only unusual feature of the semantics is the inclusion of a special element `fatal-error` 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, a softly typed language is a dynamically typed language that provides feedback on *possible* type errors. From the perspective of a compiler-writer, it is statically typed language because only type correct programs are compiled and executed.

For a soft type system to be effective and practical, it must satisfy the following two criteria:

Minimal-Text-Principle The system should not require the programmer to declare the types of any program phrases or operations.

Minimal-Failure-Principle The system should leave "most" program components unchanged unless they can produce type errors during execution.

To satisfy the minimal-failure condition, a soft type system must accommodate *parametric polymorphism*, an important form of modularity 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 α, `reverse` maps the type $\mathsf{list}(\alpha)$ to $\mathsf{list}(\alpha)$. 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 elements. Otherwise, we will not be able to type check subsequent operations on the output elements. For this reason, a soft typing system must be able to express the fact that `reverse` has type $\forall\alpha.\mathsf{list}(\alpha) \rightarrow \mathsf{list}(\alpha)$ as in ML.

A type system based only on parametric polymorphism, however, still does not satisfy minimal-failure criterion. There are two important classes of program expressions that commonly occur in dynamically typed programs that do not type-check in languages that support only parametric polymorphism.

3.1 Non-uniformity

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 *(Non-uniform if)* Let f be the function

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

where `if-then-else` has type $\forall\alpha\ \mathsf{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$.

f takes a simple 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 non-uniform expressions, we need to introduce *union* types. The union type $\alpha \cup \beta$ is the union of the sets α and β. It is a fundamentally different type construction than the *disjoint union* type construction found in many languages (including ML). The *disjoint union* of α and β forms a new set disjoint from α and β; the elements of new set are created by "tagging" the elements of α and β.

In a type system with union types, the function in Example 1 has type $\mathsf{bool} \rightarrow \mathsf{int} \cup \mathsf{nil}$. Union types also enable us to make finer grained type distinctions. As an illustration of this effect, consider the type list. It is the union of two different constructor types: the empty list `nil` and the constructed lists of the form (`cons` α L). The division of the type $\mathsf{list}(\alpha)$ into the union of the types nil and $\mathsf{cons}(\alpha)$ is important because the selectors `head` and `tail` functions produce run-time errors when applied to the empty list! Languages without union types typically define the type of `head` as $\mathsf{list}(\alpha) \rightarrow \alpha$. In this case, (`head nil`) will pass the type checker, even though it will generate a run-time error when it is executed.

3.2 Recursivity

The other troublesome class of 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 α. 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.

The true power of recursive typing, however, lies in its use with union types. The following simple example (taken from an introductory Scheme course[10]) is a good illustration of this phenomenon.

> **Example 2** Let the function `deep` be defined by the equation
>
> $$\texttt{deep} = \lambda n.(\texttt{if}\ (=?\ n\ \texttt{zero})\ \texttt{zero}\ (\texttt{cons}\ (\texttt{deep}\ (\texttt{pred}\ n))\ \texttt{nil}))$$
>
> where `zero` is the 0-ary constructor denoting 0 and $\texttt{pred} : \mathsf{suc} \rightarrow \mathsf{zero} \cup \mathsf{suc}$ is the standard predecessor function.

This function takes a non-negative integer n as input, and returns 0 nested inside n levels of parentheses. It does not type check in ML because the output type cannot be described by a simply parametric pattern or cyclic pattern. In a type system with parametric types, union types, and recursive types, `deep` has type $(\mathsf{zero} \cup \mathsf{suc}) \rightarrow \texttt{fix}\ t.\mathsf{zero} \cup \mathsf{cons}(t)$

4 A Type Language Suitable for Soft Typing

Our type language is parameterized by the set of data constructors in the programming language. We assume that the data constructors are disjoint: no value v can be generated by two different constructors. For each data constructor `c`, we define a type constructor with the same name `c`. The type language also includes 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` arity 0. The special type constructor $\rightarrow$ has arity 2.

Given a set of type constructors C, we define the set of *constructor* type terms $\Sigma(C, V)$ over a set of variables V to be the free term algebra over C, V. We define the *raw* type terms over C and V as the set that is inductively defined by the equation

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

where v is any type variable in V, c is any type constructor in C, and $\ldots$ is a list of n raw type terms where n is the arity of c.

The intended meaning of raw type terms is the obvious one. Every closed type term denotes a set of data objects. For each data constructor $c \in C$, the corresponding type construction $c(\alpha_1, \ldots, \alpha_m)$ denotes the set of all data objects of the form $c(x_1, \ldots, x_n)$ where (i) each polymorphic argument x_i in the list $x_1, \ldots, x_n$ is taken from the type denoted by the matching[1] argument α_j and (ii) each non-polymorphic argument x_i is taken from the type 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 α into β. The $+$ operator denotes the union operation on sets and the `fix` operator denotes the least fixed-point operation

[1] Type argument α_i corresponds to the ith *polymorphic* argument of the data constructor c. Hence, j may be greater than i.

on type functions. A formal definition of the semantics of type terms based on the ideal model[15, 14] of MacQueen, Plotkin, and Sethi is given in the full paper.

For technical reasons, we must impose some modest restrictions on the usage of the `fix` and $+$ operators in type terms. The set of *tidy* type terms over C and V is the set of raw type terms over C and V that satisfy the following two constraints:

1. Every subterm of the form $\texttt{fix}\ v.t$ 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 the full paper. The expressions $\texttt{fix}\ \alpha.\alpha$ and $\texttt{fix}\ \alpha.\alpha + \mathsf{n}il$ are not formally contractive, but $\texttt{fix}\ \alpha.\alpha \rightarrow \beta$ and $\texttt{fix}\ \alpha.\mathsf{nil} + \mathsf{cons}(\alpha)$ are.

2. Every subterm of the form $u+v$ is *discriminative*. A raw type term 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 construction) of $u+v$. The terms $t+\mathsf{true}$ and $\mathsf{cons}(x)+\mathsf{cons}(y)$ are not discriminative. but $\mathsf{false}+\mathsf{true}$ and $\mathsf{nil}+\mathsf{cons}(y)$ are.

Let C be the set of type constructors for **Exp**. Given a set of type variables V, the *soft type language* for **Exp** is the set of *tidy type terms* over C and V.

Our soft type system is specifically designed to facilitate reasoning about subtypes of unions, an essential characteristic of soft type systems that we identified in Section 3. To reason about subtypes, we use the following inference system patterned after a system developed by Amadio and Cardelli[3]

Definition 2 *(Inference Rules for Subtyping)*

$$\begin{array}{rcl}
\text{REFL:} & \vdash T \subseteq T & \\
\text{UNI:} & \vdash T_1 \subseteq T_1 + T_2 & \\
\text{FIX1:} & \vdash \texttt{fix}\ x.T = T[x \leftarrow \texttt{fix}\ x.T] & \\
\text{TRANS:} & \dfrac{\vdash T_1 \subseteq T_2 \qquad\qquad \vdash T_2 \subseteq T_3}{T_1 \subseteq T_3} & \\
\text{CON:} & \dfrac{\vdash S_1 \subseteq T_1 \ldots \vdash S_n \subseteq T_n}{\vdash c(S_1,\ldots,S_n) \subseteq c(T_1,\ldots,T_n)} & c \text{ an } n\text{-ary constructor, } c \neq \rightarrow \\
\text{FUN:} & \dfrac{\vdash T_3 \subseteq T_1 \qquad\qquad \vdash T_2 \subseteq T_4}{\vdash T_1 \rightarrow T_2 \subseteq T_3 \rightarrow T_4} & \\
\text{FIX2:} & \dfrac{t_1 \subseteq t_2 \vdash T_1[x \leftarrow t_1] \subseteq T_2[x \leftarrow t_2]}{\vdash \texttt{fix}\ x.T_1 \subseteq \texttt{fix}\ x.T_2} &
\end{array}$$

The inference rules assume that $+$ operator is associative, commutative, and idempotent.

The final step in defining our type language is adding the notion of universal quantification. As in ML[8, 7], we restrict universal quantification to the "outside" of type terms. In this restricted setting, type terms containing quantifiers are called *type schemes.* A type scheme is defined by the grammar:

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

where τ denotes the set of tidy terms. The types of polymorphic operations are given by type schemes rather than type terms.

The semantics for tidy type terms is easily extended to accommodate type schemes. The technical details appear in the full paper.

5 Type Inference

To assign types to programming language expressions, we use a type inference system similar to the type inference system for ML. The inference system uses both the subtype relation $\subseteq$ defined by the subtype inference system given in Section 4 and the *generic instance* relation $\leq$ defined by Damas and Milner[7] for the ML type system. A generic instance of type scheme τ is a type scheme τ' obtained by substituting tidy type terms for *quantified* variables of τ. For example, given the type $\forall\alpha.\mathsf{cons}(\alpha) \rightarrow \alpha$, then $\mathsf{cons}(\mathsf{int}) \rightarrow \mathsf{int}$ is a generic instance.

The inference rules below are the original Milner rules augmented by the single rule SUB

Definition 3 *(The soft type system rules)* Let e be a program expression and let t be a type scheme. The formula $e : t$ is called a *typing judgment.* The intended meaning is that e has type t.

Let A be a set of type assumptions of the form id : σ.

TAUT:	$A \vdash x : \sigma$	$A(x) = \sigma$
INST:	$\dfrac{A \vdash x : \sigma}{A \vdash x : \sigma'}$	$\sigma' \leq \sigma$
GEN:	$\dfrac{A \vdash e : \sigma}{A \vdash e : \forall\alpha\sigma}$	α not free in A
ABST:	$\dfrac{A \cup \{x : \tau'\} \vdash e_1 : \tau}{A \vdash \lambda x.e_1 : \tau' \rightarrow \tau)}$	
APP:	$\dfrac{A \vdash f : \tau_1 \rightarrow \tau_2 \quad A \vdash e : \tau_1}{A \vdash (f\ e) : \tau_2}$	
LET:	$\dfrac{A \vdash e_1 : \sigma \quad A \cup \{x : \sigma\} \vdash e_2 : \tau}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$	
SUB:	$\dfrac{A \vdash e : \tau}{A \vdash e : \tau'}$	$\tau \subseteq \tau'$

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 3 *(A non uniform deduction)* Let $f_1 : (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ and $f_2 : a + b \rightarrow a$ be program functions. We can immediately deduce that $(f_1\ f_2) : a \rightarrow a$. Alternatively, $A \vdash (f_1\ f_2) : a + b \rightarrow a + b$. These two typings for $(f_1\ f_2)$ are incomparable, so neither is best. Furthermore, no other type that has both types as supertypes can be derived, so there exists no best type.

Nevertheless, our type system retains the most important property of the ML type inference system, namely the soundness property and its corollaries. The soundness property simply asserts that every provable typing judgment is true, according to semantics that we have defined for the programming language, the type language, and type assertion language. This result is formally stated in the full paper. 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 and for every parameter to a type constructor. This encoding was developed by Rémy[19] to reduce inheritance polymorphism to parametric polymorphism.
- Second, we use circular unification (as in CAML[22]) instead of regular unification to solve systems of type equations. Circular unification is required to infer recursive types.

The Rémy encoding is based on the observation that set of data constructors C is *finite*. Consequently, we can enumerate (up to substitution instances) all the tidy subtypes and supertypes of a given type expression. Consider the following example. Assume that the only constructors in our type language are $a(x), b, c$. Then the supertypes of the type b are $\{b, (a(x)+b, b+c, a(x)+b+c\}$. 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 $b+c$ instead. The subtypes of $b+c$ are $\{b, c, b+c\}$. 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.

Rémy expresses these patterns in the framework of parametric polymorphism by using a single highly polymorphic type constructor $\mathcal{R}$ and two type constants $+$ and $-$ signifying "must" and "must-not" respectively. $\mathcal{R}$ has a type parameter for each type constructor (which must be instantiated by either $+$ or $-$) and a type parameter for each type argument taken by a constructor (which must be instantiated by an instance of $\mathcal{R}$). Using this notation, we can rewrite the preceding patterns as follows. The supertypes b are given by the Rémy type expression $\mathcal{R}(\tau_1, x, +, \tau_2)$. The subtypes of $b + c$ are given by $\mathcal{R}(-, x, \tau_3, \tau_4)$

The key property of this encoding is that it *eliminates* the union operator $+$ and reduces the subtyping relation to the instantiation of the polymorphic type constructor $\mathcal{R}$. As a result, the ML type assignment algorithm can be used to solve type equations. If we extend Rémy's notation to include the `fix` operator, then we can express all tidy types using Rémy's notation. More precisely, we can map any tidy type term τ to a Rémy type term $R_+(\tau)$ denoting all of the supertypes of τ. Similarly, we map any tidy type term τ to a Rémy type term $R_-(\tau)$ denoting all of the subtypes of τ. The encoding is non-trivial because we must treat the function type constructor $\rightarrow$ as a special case to cope with the fact $\rightarrow$ is anti-monotonic in its first argument.

Before we define the mapping from tidy type terms to Rémy notation, we need to introduce some subsidiary definitions. Let $||$ denote the *syntactic concatenation operator* defined by the equation:

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

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

We define the mappings R_+ and R_-] from tidy type terms to Rémy type terms as follows:

$$\begin{array}{rcl}
R_{+,i}(t) & = & \left\{ \begin{array}{ll}
t & t \in V \\
\langle +, R_+(t_1), \ldots, R_+(t_m)\rangle & c_i(t_1, \ldots, t_m) \subseteq t \\
 & c_i \neq \rightarrow \\
\langle +, R_-(t_1), \ldots, R_+(t_m)\rangle & t_1 \rightarrow t_2 \subseteq t \\
\langle \kappa_j, x'_1, \ldots, x'_m\rangle & c_i \not\subseteq t \\
\mathtt{fix}\ m.R_{+,i}(t') & t = \mathtt{fix}\ m.t'
\end{array} \right. \\
R_{-,i}(t) & = & \left\{ \begin{array}{ll}
t & t \in V \\
\langle \kappa_j, R_-(t_1), \ldots, R_-(t_m)\rangle & c_i(t_1, \ldots, t_m) \subseteq t \\
 & c_i \neq \rightarrow \\
\langle \kappa_j, R_+(t_1), R_-(t_2)\rangle & t_1 \rightarrow t_m \subseteq t \\
\langle -, x'_1, \ldots, x'_m\rangle & c_i \not\subseteq t \\
\mathtt{fix}\ m.R_{-,i}(t') & t = \mathtt{fix}\ m.t'
\end{array} \right. \\
R_+(t) & = & \mathcal{R}(R_{+,1}(t) || R_{+,2}(t) \ldots || R_{+,n}(t)) \\
R_-(t) & = & \mathcal{R}(R_{-,1}(t) || R_{-,2}(t) \ldots || R_{-,}(t))
\end{array}$$

where each occurrence of κ_j denotes a fresh type variable, distinct from all other type variables.

The inverse transformations are similar and will not be shown here. However, we note that some $\mathcal{R}$-terms correspond to a *set* of tidy terms, rather than a single term.

Our type assignment algorithm consists of three steps:

1. Encode the types of primitive operations in Rémy notation.
2. Perform ordinary type assignment via algorithm W (using circular unification).
3. Translate the Rémy type expressions back into tidy terms.

In the full paper, we will prove that our type assignment algorithm is sound.

6.1 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 $\textsf{true} + \textsf{false} \rightarrow \textsf{suc} + \textsf{nil}$ to $\texttt{mixed}$.

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 4 *(Tautology example)*

$$
\begin{aligned}
\texttt{taut} = \lambda B.&\texttt{case}\ B\ \texttt{of}\\
&\quad \texttt{true}:\quad \texttt{true}\\
&\quad \texttt{false}:\quad \texttt{false}\\
&\quad \texttt{fn}:\ ((\texttt{and}\ (\texttt{taut}\ (B\ \texttt{true})))\ (\texttt{taut}\ (B\ \texttt{false})))
\end{aligned}
$$

For the `taut` function, our algorithm produces the typing $\texttt{taut} : \beta \rightarrow (\textsf{true}+\textsf{false})$ where $\beta = \texttt{fix}\ t.(\textsf{true} + \textsf{false} + ((\textsf{true} + \textsf{false}) \rightarrow t))$.

7 Inserting run-time checks

As we explained in the introduction, *no* sound type checker can pass all "good" programs. Our type checker described in section 6 succeeds in inferring types for most programs. Nevertheless, some "good" programs will not pass that type checker. For example:

Example 5 *(Function with no type)*

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

The type checker fails because it infers incompatible constraints for the type of the argument f. The `if` test forces $f : \texttt{true} \rightarrow \texttt{true} + \texttt{false}$. Similarly, the `true` arm of the `if` requires $f : \texttt{suc} \rightarrow \texttt{z} + \texttt{suc}$. 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 6 *(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 $N_2 : (\mathsf{true} + \mathsf{suc} \rightarrow \mathsf{true} + \mathsf{false}) \rightarrow \mathsf{true} + \mathsf{false}$. But the application $N_2(\lambda\ x.x)$ is also well defined, but does not type check.

In both examples, the program is meaningful, but the type analysis is not sufficiently powerful to assign an appropriate type. A static type system rejects these programs, in spite of their semantic content. A soft type system, however, *cannot* reject programs. It must insert explicit run-time checks instead.

The support the automatic insertion of explicit run-time checks, we force the programming language to include a collection of *exceptional values* in the data domain and a collection of functional constants to the programming language called *narrowers*. The narrowers perform run-time checks and exceptional values propagate the information 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 simply the outermost constructor in the representation of the value. For a function, the primitive type is simply $\rightarrow$. 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 $t_1 + \ldots + t_n$.

There is one exceptional value for each simple type S; it is denoted error_S. Similarly, there is a narrower $\downarrow_T^S$ for every pair of simple types such that $S \subseteq T$. It is defined by the equation

$$\downarrow_T^S(v) = \begin{cases} v & v \in T \\ \mathsf{error}_T & v \in S - T \\ \texttt{fatal-error} & \text{otherwise} \end{cases}$$

The type associated with the narrower $\downarrow_T^S$ 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 sequence of steps:

1. Encode all type assumptions about primitive operations using R_+ and convert all "$-$" flags to fresh type variables—eliminating all negative information.

2. Apply algorithm W (with circular unification!) on this transformed program to construct a "positive" type for every program expression.

3. For each occurrence of a primitive, unify the "positive" type with original type (using Rémy encodings). If unification fails, insert the narrower required for unification.

In the full paper, we prove that the narrower insertion algorithm preserves the meaning of programs and that it always succeeds in constructing a type correct program. Both proofs are straightforward.

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 c_1 and c_2 denote the narrowers $\downarrow^{\mathsf{z+suc+true+false}}_{\mathsf{z+suc}}$ and $\downarrow^{\mathsf{z+suc+true+false}}_{\mathsf{true+false}}$, respectively.

The insertion algorithm changes the function N_1 in example 5 to

$$N_1' = \lambda\ f.\ \texttt{if}\ \mathsf{c}_1(f(\texttt{true}))\ \texttt{then}\ \mathsf{c}_1(f(5)) + \mathsf{c}_1(f(7))\ \texttt{else}\ f(7)$$

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

Similarly, in example 6, 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

$$N_2' = \lambda\ f.\ \texttt{if}\ \mathsf{c}_1(f(\texttt{true}))\ \texttt{then}\ f(5)\ \texttt{else}\ f(7)$$

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

$$(\mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false} \rightarrow \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false}) \rightarrow \mathsf{z} + \mathsf{suc} + \mathsf{true} + \mathsf{false}$$

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

8 Related Work

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

More recently, Thatte[20] has developed the notion of "quasi-static typing" that augments a static type system with a general type Ω. Thatte's system resembles soft typing in that it insert narrowers when necessary, ensuring that all programs can be executed. However, it does not meet the other criteria required for soft typing. It requires explicit declarations of the argument types for functions, yet 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.

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. Two of the most prominent examples are the systems developed by Aiken and Murphy[2] and Cohagan and Gateley[6]. These systems, however, were not 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. In the arena, the work of Mitchell[18], Fuh-Mishra[11, 12, 13], and, of course Rémy[19] is particularly noteworthy and has exerted a strong influence on our work.

9 Conclusions

The principal contribution of this research is the introduction of a new paradigm for program typing called *soft typing* that combines the best features of *static* and *dynamic* typing. The principal technical contributions include

- A type system specifically designed to assign types to a large fraction of programs written in dynamically typed languages. The type system incorporates union types, recursive types, and parametric polymorphism. The type system includes a sound type inference system for deducing types for program expressions.
- A simple yet powerful algorithm for automatically assigning types to program expressions, without the need for explicit type declarations.
- A simple algorithm for frugally inserting explicit run-time checks in programs for which the type assignment algorithm fails. Consequently, *any* program may be safely executed.

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] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, August 1990. (To Appear).

[4] Andrew W. Appel and David MacQueen. Standard ML of new jersey reference manual. (in preparation), 1990.

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

[6] William Cohagan and John Gateley. Interprocedural dataflow type inference. Submitted to SIGPLAN '91, 1990.

[7] 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.

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

[9] 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. (To Appear).

[10] D. P. Friedman and M. Felleisen. *The Little Lisper*. Science Research Associates, third edition, 1989.

[11] You-Chin Fuh. *Design and Implementation of a Functional Language with Subtypes*. PhD thesis, State University of New York at Stony Brook, 1989.

[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] 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.

[15] 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.

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

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

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

[19] 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.

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

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

[22] Pierre Weis. *The CAML Reference Manual*. INRIA, 1987.