Combining Strict and Soft Typing in Functional Programming

Manfred Widera, Christoph Beierle
Fachbereich Informatik, FernUniversität Hagen, 58084 Hagen
email: {Manfred.Widera, Christoph.Beierle}@FernUni-Hagen.de

Abstract. We discuss the properties of strictly typed languages on the one hand and soft typing of the other and identify disadvantages of these approaches to type checking in the context of powerful type languages. To overcome the problems we develop an approach that combines ideas of strict and soft typing. This approach is based on the concept of complete typing that is guaranteed to accept every well-typed program. The main component of a complete type checker is defined.

1 Introduction

Types are a component of many higher programming languages following different programming paradigms, e.g. C [8], C++ [11], Standard ML [13] and Haskell [6]. In functional programming there are two commonly used approaches to type checking:

- In dynamically typed languages like, e.g. Scheme [7] every data object carries a type tag that is used for type checking at runtime.
- Statically typed languages like ML and Haskell use type inference for type checking at compile time. These languages reject every program that cannot be proven to be well-typed.

Types normally express sets of values with common properties. Early languages focussed on system dependent common properties, e.g. a common internal representation with a fixed size. Recent functional programming languages on the other hand focus on the programmer's view of types and allow to define new types that have common properties from the programmer's point of view (e.g. Standard ML [13] or Haskell [6]). A very exact and powerful type language could be helpful to the programmer in detecting errors, but unfortunately too powerful type languages can cause problems for sound type systems. They tend to force a type checker to reject too many programs that should indeed be accepted.

A further approach in dynamically typed languages, called soft typing (see e.g. [3]), employs static type checking in order to identify function calls that need a runtime check because they might be ill-typed. Runtime checks for calls that can be statically proven to be well-typed can be dropped. In contrast to statically typed languages soft typing has the disadvantage that it does not reject *any* programs. If the programmer ignores the type warnings this may result in runtime errors. Furthermore, for every warning the programmer has to decide whether the warning results from a type error or from a weakness of the type checker.

In this paper we propose an approach to type checking that tries to avoid these weaknesses of both sound, but too strict and soft typing approaches in the context of powerful type languages. We introduce the concept of *complete type checking* that is guaranteed to accept every well-typed program. It also extends soft typing by rejecting programs that cannot be executed properly without generating a runtime error. Our type checking and inferencing method supports powerful type languages including subtyping and is applicable to dynamically typed languages like Scheme.

The paper is organized as follows: In Sec. 2 we give a motivation for complete type checking and explain the benefits of combining complete typing with soft typing. Section 3 defines the main component of a complete type checker and summarizes its main properties. Section 4 gives a summary of related work. Section 5 contains some conclusions.

2 The Use of Complete Type Checking

2.1 Disadvantages of Sound Type Checking

The usual approach to type checking is *sound* type checking that is used either in strongly typed languages or as a soft type checker in dynamically typed languages. Soundness of a type checker means that it accepts just programs that cannot generate a runtime type error. In other words sound type checkers follow Milner's slogan [9] "Well-typed programs cannot go wrong".

No matter in which way sound type checking is used the expressiveness of the type language must be restricted in order not to reject too many correct programs. Such a situation is given in Ex. 1:

Example 1. Consider the following function definition.

```
(define (with-div x y)
  (/ x (f y)))
```

Suppose f is an arbitrary function with result type `num` and 0 is not part of the value set of f. Suppose further that there is a sound type checker and that the type system can express the type of all numbers excluding 0. We normally cannot prove that f does not yield zeros for the whole set of possible inputs for y (see e.g. [12]). Thus, we cannot prove *with-div* free of type errors. □

The example shows a program that cannot go wrong, but is ill-typed with respect to the sound type checker. This can cause the following consequences:

- In a strongly typed language we observe that programs that cannot go wrong, but are not well-typed with respect to the type checker are rejected. By increasing the expressive power of the type language and the exactness of the typings for predefined functions we must expect the number of such correct but rejected programs to increase.

– A soft typing system would have to raise a warning for a function call that is indeed well-typed. When the number of such warnings on well-typed (with respect to runtime type errors) calls increases the system provides less help to the programmer to find real type errors quickly.

A further problem for soft typing is shown by the following example Ex. 2:

Example 2. Consider the following erroneous implementation of the function reverse and its use:

```
 1   (define (reverse l)
 2     (if (null? l)
 3         '() ; reversed empty list is empty
 4       (append (reverse (cdr l))
 5                 ; reverse rest
 6               (car l))))
 7                 ; append first element at the end.

 8  (define (generate n)
 9    (if (= 0 n) ()
10        (cons n (generate (- n 1)))))

11  (define (f n)
12    (reverse (generate n)))
```

There is an error in the second argument (line 6) of the call to the predefined function *append* (lines 4-7) because from the call to *reverse* in f (line 12) it can be inferred that (*car* l) in line 6 is not a list in every case. But although there is a call that must go wrong in the given context the soft typing system does not reject the program. □

As this example shows soft typing reacts "too soft" on real *provable* type errors. Altogether, the user would have to check a lot of warnings of a soft typing system with a powerful type language in order to detect a single type error.

2.2 Motivating the New Approach

The following properties of functional programs are needed to explain the completeness of a type checker:

Let f be a predefined function and $dom(f)$ the set of input values f is applicable to. A *misapplication* of f is a call $(f\ a)$ where $a \notin dom(f)$.

Let P be a functional program and e an expression in P. e *(always) goes wrong* if every evaluation of e causes a misapplication of some predefined function f.[1] The program P *goes wrong* if P contains an expression e that goes wrong.

An expression e in a program P *conditionally goes wrong* if an execution path in P starting at e leads to the misapplication of a predefined function. P *conditionally goes wrong* if an expression e in P conditionally goes wrong.

[1] We assume the functional language to be strict and to use eager evaluation.

Example 3. In the program of Ex. 2 the call to *reverse* in f conditionally goes wrong because of the execution path to *append* in *reverse*.

An example for a call that (always) goes wrong (with respect to the program in Ex. 2) is $(f\ 3)$; please note that the call $(f\ 0)$ does not cause a misapplication because the else-part in *reverse* containing the ill-typed *append*-call is never reached. Another example for a call that goes wrong is $(*\ 'a\ 3)$ because the first argument of $*$ is not a number. □

By completeness of a type checker we mean the following property: If a program P does not go wrong then P is not rejected by the type checker.

A complete type checker circumvents the problem of a strongly typed language to reject programs that cannot go wrong. On the other hand it is not as weak as a soft typing system because the complete type checker can reject provably ill-typed programs.

The combination of soft typing with complete typing divides the output messages of the system into *errors* that cause the rejection of the program and therefore must be corrected, and *warnings* that mark calls which could not be proven to be well-typed, but are not *provably* wrong. This structure of output messages has a number of further advantages for the programmer both in debugging programs and in proving certain correctness statements:

In debugging a program one can start correcting the fatal errors of the program before taking care of the type warnings (i.e., either proving type correctness of the calls or correcting them). By getting the output messages structured into errors and warnings the programmer is guided through the increased number of calls that are not provably well-typed due to a more powerful type language. In no case the program has to be changed just to satisfy the type checker.

One common argument for strongly typed languages is the fact that one gets a partial correctness proof for every accepted program. On the one hand by the absence of type *errors* the complete type checker yields such a prove automatically for at least a subset of type errors. On the other hand type *warnings* spot on all those calls for which a correctness prove could not be generated automatically. When the programmer proves the necessary properties of these calls the resulting prove yields stronger results than the type checker of a strongly typed languages without restricting the expressive power of the language.

2.3 Realizing a complete type checker

This section explains the intended behaviour of a complete type checker and how it can be achieved. It motivates the definitions given in Sec. 3.

In powerful type languages the problem of type inference is often undecidable. When the exact type of an expression cannot be inferred the usual approach is to infer a supertype, i.e. a type that covers all values that are of the wanted exact type. Let τ be the type inferred for the argument of a function call, σ the expected input type of the called function and let $\sqsubseteq$ denote the subtype relation. Then a sound type checker with subtyping facility checks the call for an approximation of the property $\tau \sqsubseteq \sigma$. The type checker will accept the program if all calls fulfill

this property and it will reject it if one call does not. However, it is possible that only the additional values that are covered by τ but not by the exact type of the argument cause the test $\tau \sqsubseteq \sigma$ to fail and the program to be rejected (c.f. Ex. 1 with $\sigma = \texttt{num}$ and τ the type of all numbers without 0).

For complete type checking we do not want a program to be rejected just because of additional values in the inferred type of an argument. Since we cannot distinguish the values of the exact type from those additionally in the inferred type the type checker should reject only those calls that must go wrong for every value of the inferred type. Therefore, the complete type checker tests every call in the program for the property $\tau \cap \sigma \neq \emptyset$. Every call that does not fulfill this property is caused by an expression that conditionally goes wrong. A formal definition of a system that is complete with respect to the definition of conditionally going wrong is given in Sec. 3.

3 The Definition of Complete Subtyping

In this section we define the main component of a complete type checker for a usual higher order functional language (one can think e.g. of Scheme or Standard ML with dynamic typing). We essentially just make use of:

- lambda abstraction.
- function application.
- the existence of tuple like data constructors like *cons* for pairs.
- the existence of a set $\mathcal{D}$ of predefined functions f with given sets $D(f)$ of typings where every given output type contains all values that could occur with the corresponding input type.

The set of all values expressible in this language will be denoted by $\mathcal{V}$.

Definition 1 (type language). *Type expressions are given by a type language consisting of:*

1. *A finite set $\mathcal{B} = \{\bot, \top, b_1, \ldots, b_k\}$ of type constants called base types.*
2. *A finite set $\mathcal{C} = \{\rightarrow, \cup, \cap, \mu, c_1, \ldots, c_n\}$ of type constructors with arity ≥ 1.*

Every finite term t generated from the type constants, type constructors and a set of type variables as usual is a type expression. We call a type expression ground if it does not contain a free (i.e. not bound by μ) type variable.
The set of all type expressions (or types for short) is denoted by $\mathcal{T}$. $\mathcal{T}_G \subset \mathcal{T}$ denotes the set of all ground type expressions.

For a simpler representation we omit the use of types with free variables in inferred types. In the sets $D(f)$ for $f \in \mathcal{D}$ free variables are considered all-quantified.

Definition 2 (semantics of types). *Let $t \in \mathcal{T}_G$ be a ground type expression and let $\mathcal{V}$ denote the set of all values expressible in the current functional language. Every type $t \in \mathcal{T}_G$ represents a set of values $V(t) \subseteq \mathcal{V}$:*

- $\bot$ *is the empty type just containing the value* $\bot$ *expressing non-termination.*
- $\top$ *is the type representing the set of all values.*
- *The base types* $b_1, \ldots, b_k$ *represent sets of simple values with the following property: If* $b, b' \in \mathcal{B}, V(b) \cap V(b') \neq \emptyset$ *then there exists some* $\tilde{b} \in \mathcal{B}$ *with* $V(\tilde{b}) = V(b) \cap V(b')$.
- $A \to B$ *contains all functions that map all values* $a \in V(A)$ *to values* $b \in V(B)$.
- *Types defined by* $\cup$ *and* $\cap$ *contain the union resp. intersection of the values of the element types.*
- $\mu X.t$ *with* t *containing* X *as free variable defines a recursive type as the least fixed points of the equation* $V(t') = V(t[X \leftarrow t'])$.[2]
- *In general* $c_1, \ldots, c_n$ *are free type constructors and correspond to tuple like data constructors. E.g. the type* $(Tcons\ A\ B)$ *represents the value set*

$$\{(cons\ a\ b) \mid a : A, b : B\}\,.$$

For $v \in V(t)$ *we also write* $v : t$.

Definition 3 (type hierarchy). *The subset relation* $\subseteq$ *on the power set of* $\mathcal{V}$ *introduces a type hierarchy on the set* $\mathcal{T}_G$ *of all ground types with the subtype relation* $\sqsubseteq_R$ *by the following definition:*

$$t_1 \sqsubseteq_R t_2 \iff V(t_1) \subseteq V(t_2)\,.$$

Though there are subtyping procedures for quite powerful type languages (see e.g. [4]) we are going to use approximations of the type hierarchy defined in Def. 3 with certain properties. By this we do not rely on type languages for which an algorithm deciding the semantic subtype relation exists.

Definition 4 (compatible subtype relation). *A subtype relation* $\sqsubseteq$ *is compatible to another subtype relation* $\sqsubseteq'$ *if the following condition holds:*

$$t_1 \sqsubseteq t_2 \Longrightarrow t_1 \sqsubseteq' t_2$$

$\sqsubseteq$ *is called compatible if it is compatible to* $\sqsubseteq_R$. *In the following we use the notion* $t_1 \sqsubset t_2$ *for* $t_1 \sqsubseteq t_2 \wedge t_2 \not\sqsubseteq t_1$.

The idea of complete type checking applies in particular to calls of predefined functions. When a function $f \in \mathcal{D}$ is applied to an expression e where the most special type of e and no input type of f have common elements then a conditional type error (i.e. the misapplied call to a predefined function corresponding to an expression that conditionally goes wrong) is detected. Otherwise we select all input types of f that are most special and have a maximal number of common elements with e. We type the call $(f\ e)$ with the union of the corresponding output types. This is formalized in the following definitions.

[2] $t[X \leftarrow t']$ expresses the type t with every occurence of the variable X replaced by t'.

Definition 5 (partial parameter dependent output types). *Let A be a set of type assumptions (i.e. a set of type assignments and subtype conditions on type variables), $f \in \mathcal{D}$ a predefined function, $D(f)$ the set of all given typings of f and e a value expression. A type σ is a partial parameter dependent output type (ppo) of f with respect to e and A (written $\sigma \in PPO(A, f, e)$) if the following properties hold:*

1. $A \vdash e : \tau'$.[3]
2. *For every* $\tilde{\tau}'$ *with* $A \vdash e : \tilde{\tau}'$ *we have* $\tilde{\tau}' \not\sqsubseteq \tau'$.
3. $\tau \to \sigma \in \widehat{D(f)}$ *where* $\widehat{D(f)}$ *is the set of all ground instances of types in* $D(f)$.
4. $EI(\tau, \tau')$ *is* false *where* EI *approximates the test for empty intersection of two types with the following property:*

$$EI(\rho, \rho') \Longrightarrow \rho \cap \rho' = \bot\,.$$

5. *If* $\tilde{\tau} \to \tilde{\sigma} \in \widehat{D(f)}$ *is a typing of* f *then the following holds:*
   (a) $\tau \cap \tau' \not\sqsubseteq \tilde{\tau} \cap \tau'$.
   (b) $\tau \cap \tau' = \tilde{\tau} \cap \tau' \Longrightarrow \tilde{\tau} \not\sqsubseteq \tau$.

Remark 1. Informally, for the most special type τ' of e (1),(2) Def. 5 chooses the most special input type τ of f (3), (5b) that causes the least possible restriction of the intersection $\tau \cap \tau'$ (5a). The calculation fails in the case of an empty intersection $\tau \cap \tau'$ (4). The resulting partial parameter dependent output type is the output type of f corresponding to the chosen input type.

Note that (4) formalizes the test whether f is applicable to e. The other conditions are needed to get exact result types.

Though we have defined $PPO(A, f, e)$ just for unary function symbols f the definition is easy to extend by introducing tuples via a type constructor $\times$ defined by $V(A \times B) := V(A) \times V(B)$ and a corresponding data constructor $(\cdot, \cdot)$. This is shown in Ex. 4.

Example 4. Consider a function application $(+\ x\ y)$ and a set $D(+)$ of typings for $+$:

$$\texttt{nat} \times \texttt{nat} \to \texttt{nat} \in D(+) \tag{1}$$

$$\texttt{int} \times \texttt{int} \to \texttt{int} \in D(+) \tag{2}$$

$$\texttt{num} \times \texttt{num} \to \texttt{num} \in D(+) \tag{3}$$

$$\texttt{string} \times \texttt{string} \to \texttt{string} \in D(+) \tag{4}$$

(1), (2) and (3) denote the sum of naturals, integers and arbitrary numbers, respectively, with $\texttt{nat} \sqsubseteq \texttt{int} \sqsubseteq \texttt{num}$ and (4) denotes the concatenation of strings.

Suppose furthermore that the most special type with respect to A is $\texttt{nat}$ for x and $\texttt{int}$ for y. Then for calculating $PPO(A, +, (x, y))$ we have:

- $\tau' := \texttt{nat} \times \texttt{int}$.
- The intersection $\tau \cap \tau'$ is most general for $\tau \in \{\texttt{int} \times \texttt{int}, \texttt{num} \times \texttt{num}\}$.
- Because of $\texttt{int} \times \texttt{int} \sqsubseteq \texttt{num} \times \texttt{num}$ we have $\tau := \texttt{int} \times \texttt{int}$.

[3] The notion $A \vdash e : \rho$ is defined in Fig. 1.

$$\frac{}{A \cup \{x:\tau\} \vdash x:\tau}\ (Var) \qquad \frac{A \vdash e:\tau}{A \vdash e:\tau'}\ \tau \sqsubseteq \tau'\ (Sub)$$

$$\frac{A \cup \{x:\tau'\} \vdash e:\tau}{A \vdash \lambda x.e:\tau' \to \tau}\ (Abs) \qquad \frac{A \vdash e:\tau', A \vdash f = \lambda x.e':\tau' \to \tau}{A \vdash (f\ e):\tau}\ (App\text{-}Lambda)$$

$$\frac{A \vdash e:\tau'}{A \vdash (f\ e):\tau} \quad \begin{array}{l} f \in \mathcal{D} \\ \tau := PO(A,f,e) \neq \bot \end{array} \qquad (App\text{-}Pre)$$

Fig. 1. Type inference rules for a simple complete type checker CC

– The resulting unique ppo is `int`.

If the most special types of x and y are $\texttt{nat} \cup \texttt{string}$ and $\texttt{int} \cup \texttt{string}$, respectively, we get $\tau' := (\texttt{nat} \cup \texttt{string}) \times (\texttt{int} \cup \texttt{string})$ and for τ we get $\texttt{int} \times \texttt{int}$ and $\texttt{string} \times \texttt{string}$. Thus, we have the two ppos `int` and `string`.

If we have the types `nat` and `string` for x and y, respectively, then $\tau' := \texttt{nat} \times \texttt{string}$, but the definition of τ fails since there is no input type τ of $+$ with $\neg EI(\tau, \tau')$. (Note that $t \times \bot = \bot \times t = \bot$.) □

As Ex. 4 shows there can be none, one or several ppos. For no ppo a type error has been detected. In the case of several ppos the function application has to be typed with the union of them. This motivates the following definition:

Definition 6 (parameter dependent output type). *Let A, e and f be as in Def. 5. The parameter dependent output type of f with respect to e and A is defined as*

$$PO(A,f,e) := \begin{cases} \bigcup_{\sigma \in PPO(A,f,e)} \sigma & \text{if } PPO(A,f,e) \neq \emptyset \\ \bot & \text{else}. \end{cases}$$

With a compatible subtype relation $\sqsubseteq$ a simple complete (with respect to programs that conditionally go wrong) type checker is given by Fig. 1. The new idea of completeness can be found in the typing rule (*App-Pre*) for calls to predefined functions. The idea is to report exactly those calls for which the inferred argument type contains no values the function is applicable to.

The type checker presented in Fig. 1 is a basic one. Further inference rules e.g. for typing *let* or conditional types as presented in [1] should be easy to adapt.

Theorem 1 (completeness of type checking). *Let CC be a type checker based on the typing rules of Fig. 1 with a compatible subtype relation $\sqsubseteq$. When CC only reports those calls $a = (f\ e)$ with $f \in \mathcal{D}$ that cannot be typed according to* (App-Pre) *because $PO(A,f,e) = \bot$ then CC reports only calls in programs that conditionally go wrong.*

Example 5. Consider the program from Ex. 2. Typing the function f causes the typing of *reverse* with the input type $l \leftarrow (list\ \texttt{num})$, which in turn causes a message to be generated for *append*. (From the control flow information available during type inference the system can identify the call $(reverse\ (generate\ n))$ in f as conditionally going wrong with the execution path to *append*. The condition for this path to be executed is $(not\ (null?\ (generate\ n)))$.) □

One should note that the system described here does not (yet) return *error* messages. This is the case because of the following two reasons:

- An expression that goes wrong need not be executed in a program. E.g. in Ex. 2 the function f may never be called.
- An expression that is reported by our system can still be executable without a runtime error if the control reached the expression on a different path. If e.g. the program in Ex. 2 contains a call $(reverse\ '((1)\ (2)\ (3)))$ the expression $(append\ \ldots)$ will not generate a misapplication of *append* for this call.

As a result we cannot be sure whether a detected problem in the program will really cause an error. But to every reported call a we can additionally compute the corresponding call a' that conditionally goes wrong (i.e. the entry point of a path that causes a runtime error) and the error condition (i.e. the conjunction of all conditions an argument to a' must fulfill to result in a call to a). This information can be used by a further stage of the type checker to determine whether the program must be rejected. This is the case if a call (always) goes wrong, i.e. its error condition is always fulfilled. Further reasons for rejecting a program can be error conditions that are always fulfilled except of special cases like empty lists (like in Ex. 5). The motto behind this would be the idea that one is not interested in functions that go wrong in *all* cases except for special cases like the empty list.

4 Related Work

This work uses the idea of type inference with subtyping [2], [10]. As we have shown in Ex. 1 undecidable problems prevent strongly typed languages with too powerful type hierarchies from being useful.

The idea of soft typing as presented in [3] can help to establish subtyping with a powerful type hierarchy. Besides reducing the number of runtime type checks as described in [15] soft typing allows to build tools that generate warnings to spot on potentially erroneous program parts. An example of set based analysis with output very similar to soft typing is MrSpidey [5] working under DrScheme.

A first approach to overcome the disadvantage of soft typing not to reject any programs is given in [14], but it depends strongly on a restricted type language. By the work presented here it is possible to build powerful type checkers with subtyping that benefit from both strongly typed languages and soft typing.

5 Conclusion and Future Work

In this paper we have motivated a new approach to type inference with subtyping that puts the focus on completeness of the type checker and we have defined the first component of such a type checker. By accepting every well-typed program and rejecting only those programs that are provably ill-typed we can use more powerful type languages without restricting the set of accepted programs due to

inaccuracies of the type inference system. The programmer on the other hand gets detailed support in detecting errors. Further warnings of an additional sound soft typing system can spot on those parts of the program that could not be proven to be well-typed.

A further component that is needed for a complete type checker analyzes calls that conditionally go wrong in order to prove a sufficient condition to reject the program. Our current work includes the formal definition of conditions that make a program *unacceptable* and an appropriate check for these conditions.

References

1. A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, Jan. 1994.
2. A. S. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming and Computer Architecture*, pages 31–41. ACM Press, June 1993.
3. R. Cartwright and M. Fagan. Soft typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.
4. F. M. Damm. Subtyping with union types, intersection types and recursive types. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software: International Symposium*, volume 789 of *Lecture Notes in Computer Science*, pages 687–706. Springer-Verlag, Apr. 1994.
5. C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 23–32, Philadelphia, Pennsylvania, 21–24 May 1996. *SIGPLAN Notices* 31(5), May 1996.
6. P. Hudak, J. Peterson, and J. H. Fasel. *A Gentle Introduction to Haskell – Version 1.4* –, Mar. 1997.
7. R. Kelsey, W. Clinger, and J. R. (Editors). *Revised^5 Report on the Algorithmic Language Scheme*, Feb. 1998.
8. B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition, ANSI C.* Prentice Hall, 1988.
9. R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, Dec. 1978.
10. G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, Dec. 1994.
11. B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison Wesley, 1997.
12. P. S. Wang. The undecidability of the existence of zeros of real elementary functions. *Journal of the ACM*, 21(4):586–589, Oct. 1974.
13. Å. Wikström. *Functional Programming using Standard ML.* Prentice Hall, 1987.
14. A. K. Wright. *Practical Soft Typing.* PhD thesis, Rice University, Houston, Texas, Aug. 1994.
15. A. K. Wright and R. Cartwright. A practical soft type system for scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 250–262, June 1994.