A Sketch of Complete Type Inference for Functional Programming

Manfred Widera

Fachbereich Informatik
FernUniversität Hagen
58084 Hagen
Germany
E-Mail:Manfred.Widera@Fernuni-Hagen.de

Abstract. Complete type inference for functional programming is an approach to incorporate static type inference into dynamically typed languages that is based on the following idea: For every program or program expression that can be evaluated without a runtime type error, types denoting all valid input values (in case of functions) and all corresponding output/result values are inferred. A type error is just raised for program expressions that must provably fail for every input.
In this work we summarize the presentation of a complete type checker. We motivate that complete type checking consists of a check for every function call whether the input type expected by the function and the type inferred for the argument have common elements. After sketching an algorithm that tests two types for common elements the type inference process is summarized in terms of an abstract interpretation.

1 Introduction

Type checking with a very exact and powerful type language could be helpful in detecting errors, but unfortunately too powerful type languages can cause problems for sound type systems. They tend to force the type checker of a *statically typed language* to reject too many programs that should indeed be accepted.

Soft typing for dynamically typed languages (e.g. [2]) employs static type checking in order to identify function calls that might be ill-typed. Runtime type checks for calls that can be statically proven to be well-typed can be dropped. Soft typing does not reject *any* programs. The type warnings, if ignored, may result in runtime errors. Furthermore, for every warning the programmer has to decide whether it results from a type error or from a weakness of the type checker.

In this paper we introduce the concept of *complete type checking* that is guaranteed to accept every well-typed program. It extends soft typing by rejecting programs that cannot be executed properly without generating a runtime error. Our type checker supports powerful type languages including subtyping. In detail the existing definition of complete type checking focuses on the language Scheme. It covers most of Scheme's language constructs as defined in [6]. However, the algorithms are designed general enough to be adapted to other languages easily.

The rest of the paper is organized as follows: Section 2 gives a motivation of complete type checking. In Sec. 3 we motivate and describe the differences of our type definitions over the usual approach and sketch an algorithm detecting common elements of two types. This algorithm turns out to be a main component of the complete type checker presented in Sec. 4 in terms of an abstract interpretation. Section 5 gives a conclusion.

2 Motivating Complete Type Checking

2.1 Disadvantages of Sound Type Checking

The usual type checkers are *sound* and are used either in strongly typed languages or as soft type checkers in dynamically typed languages. Soundness means to accept just programs that cannot cause runtime type errors. I.e. sound type checkers follow Milner's slogan [7] "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[1] in order not to reject too many correct programs:

Example 1. Consider the following function definition.

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

Suppose f is a 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 using a type language that can express the type of all numbers excluding 0. We normally cannot prove that f does not yield zeros for all possible inputs for y (cf. e.g. [11]). Thus, we cannot prove *with-div* free of type errors.[2]

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

[1] Alternatively, explicit type annotations must be provided which are not considered here.

[2] Because of this problem, the division by zero normally lies outside the scope of a sound type checker.

– In a strongly typed language programs that cannot go wrong, but are not well-typed with respect to the type checker are rejected. By increasing the number of different subtypes in the type language the number of correct but rejected programs might increase.
– A soft typing system raises a warning for a function call that is indeed well-typed. When the number of warnings on runtime-correct calls increases the system provides less help in finding real type errors quickly.

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

Example 2. Consider the following erroneous implementation of 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)))) ; should be (list (car l))
 7                 ; first element to 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 any 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 *provable* type errors, i.e. function calls that provably cannot succeed. Altogether, the user would normally have to check several warnings of a soft typing system with a powerful type language for every real type error (s)he detects.

2.2 Motivating the New Approach

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 $a \notin dom(f)$.

In principle by complete type checking we want to detect those function calls in a program that cannot succeed without going wrong: 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.[3] P *goes wrong* if it contains an expression e that goes wrong.

An expression e in a program P *conditionally goes wrong* if there is an execution path in P starting at e that leads to the misapplication of a predefined function whenever followed. P *conditionally goes wrong* if an expression e in P conditionally goes wrong. By the notion of conditionally going wrong we take into account the fact that the context of a function call can restrict its arguments in a way that enforces a failure. If a function call conditionally goes wrong then there exists at least one such context that must fail and therefore the program has to be rejected.

Example 3. In the program of Ex. 2 the call to *reverse* in f (line 12) 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 $(f\ 3)$ (i.e. if this program is extended by a function containing the call $(f\ 3)$ then this call goes wrong); please note that the call $(f\ 0)$ (when added to the program) does not cause a misapplication because the else-case in *reverse* containing the ill-typed *append*-call is never reached. Another example for a call that goes wrong in every program is $(*\ 'a\ 3)$ because the first argument of $*$ is not a number.

By completeness of a type checker we mean that a program P that does not (conditionally) go wrong is not rejected. A complete type checker circumvents the problem of a strongly typed language to reject programs that cannot go wrong, but it is not as weak as soft typing because it can reject provably ill-typed programs.

The combination of soft and complete typing yields both *errors* that cause the rejection of the program and *warnings* that mark calls which could not be proven to be well-typed, but are not *provably* wrong. This structure of messages has the following advantages:

When testing a program for type errors one can start correcting the errors of the program before taking care of the warnings (i.e., either proving correctness of the calls or correcting them). By the structure of 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.

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

2.3 Realizing a Complete Type Checker

In powerful type languages type inference is often undecidable. When an exact set of values of an expression cannot be inferred or cannot be expressed in the type language the usual approach is to infer a supertype, i.e. a type that covers all values that are of the desired exact type. Consider the function call $(f\ a)$. Let t be the type inferred for the argument a. A sound type checker with subtyping facility checks the call for an approximation of $\langle\!\langle t \rangle\!\rangle \subseteq \mathit{dom}(f)$.[4] It accepts the program if all calls fulfill this property and rejects it if one call does not. However, maybe just the additional values that are covered by t but cannot occur as values of a cause the test $\langle\!\langle t \rangle\!\rangle \subseteq \mathit{dom}(f)$[5] to fail and the program to be rejected (c.f. Ex. 1 with $t = \texttt{num}$ and s the type of all numbers without 0).

In complete type checking a program should not be rejected just because of additional values in an inferred argument type. Since we cannot distinguish the possible values of a from those additionally in t the type checker rejects only calls that must go wrong for every value of t. Therefore, the complete type checker tests every call in the program for $\langle\!\langle t \rangle\!\rangle \cap \mathit{dom}(f) \neq \emptyset$ (or more precisely $\langle\!\langle t \rangle\!\rangle \cap \langle\!\langle s \rangle\!\rangle \neq \emptyset$ where s is a type approximating $\mathit{dom}(f)$). Every call that does not fulfill this property is caused by an expression that conditionally goes wrong.

3 Operations on Types

3.1 The Type Language

The type language that is used by our complete type inference system is a quite powerful one that is generated as usual from base types, type variables, and free type constructors. Furthermore, union types, type difference (usable for conditional expressions), and recursive type definitions are considered.

In the following examples we will use the types `nat` and `num` for natural or arbitrary numbers, `string` for strings, `nil` for the empty list, `error` for the type denoting a type error, $(\cdot\ .\ \cdot)$ for cons pairs, $\top$ for the type of all values, and $t_1 \setminus t_2$ for the difference type denoting all values of t_1 that are not denoted by t_2.

The usual function type constructor $\rightarrow$ turns out to be inappropriate for complete type inference for the following reason: As stated in Sec. 2.3

[4] $\langle\!\langle t' \rangle\!\rangle$ is the set of all values denoted by a type t'.

[5] More precisely, a test $\langle\!\langle t \rangle\!\rangle \subseteq \langle\!\langle s \rangle\!\rangle$ where s is a type approximating $\mathit{dom}(f)$ is performed.

complete type checking tries to approximate the test

$$\langle\!\langle t \rangle\!\rangle \cap dom(f) \neq \emptyset \tag{1}$$

by a test

$$\langle\!\langle t \rangle\!\rangle \cap \langle\!\langle s \rangle\!\rangle \neq \emptyset\,. \tag{2}$$

In order not to reject well-typed programs the test (2) must succeed whenever (1) succeeds. This is just the case if s denotes all values of $dom(f)$, i.e. if $dom(f) \subseteq \langle\!\langle s \rangle\!\rangle$ holds. Since a type $s' \to t_{\text{out}}$ of f just implies $\langle\!\langle s' \rangle\!\rangle \subseteq dom(f)$, the input types given by the usual function types of f are inappropriate for the intended test of complete type checking.

For complete type inference the function type constructor is replaced as follows:
For output purposes a modified construction of function types, called I/O-representation, is needed. Such an I/O-representation for a function f is given by a set of I/O-representation pairs $IN_i \dashrightarrow OUT_i$ with types IN_i and OUT_i such that:

- $dom(f) \subseteq \bigcup_i \langle\!\langle IN_i \rangle\!\rangle$
- $\forall_i f(\langle\!\langle IN_i \rangle\!\rangle) \subseteq \langle\!\langle OUT_i \rangle\!\rangle \cup \{\texttt{error}\}$ (where `error` denotes a type error caused by applying a function to an inappropriate argument).

In contrast to the usual function type constructor every value in $dom(f)$ is denoted by at least one of the IN_i.

During type inference most functions are given by either a predefined function definition or a lambda closure. In both cases the abstract interpretation implementing the type inference process can work on abstractions of these functions directly. When a higher order function f expects a function f' as input, but f' is not known from a call to f then we can infer a type for f' that is given by a modification of an I/O-representation called PI/PO-representation.[6] This is described in detail in [14].

3.2 Common Elements of Types

As stated in Sec. 2.3 complete type inference is based on checking constraints of the following form: Given two types t_1 and t_2. Is there a common element $\perp \neq v \in \langle\!\langle t_1 \rangle\!\rangle \cap \langle\!\langle t_2 \rangle\!\rangle$? More precisely, for types t_1 and t_2

[6] The name PI/PO-representation expresses that the contained input and output types are $\underline{p}$artial, i.e. do not cover all input and output values possible for the denoted functions.

containing variables: is there a value $v \neq \bot$ and a substitution ρ such that $v \in \langle\!\langle \rho(t_1) \rangle\!\rangle \cap \langle\!\langle \rho(t_2) \rangle\!\rangle$? The answer to this question can be approximated by an algorithm *CE* (common elements) that consists of two stages:

Stage 1 (called *S-CE*) traverses the given types in a manner comparable to term unification. It collects constraints of the form $A \leftarrow t_A$ for a variable A and a type t_A which express that A must be instantiated to a type denoting at least all values denoted by t_A. t_A can be defined by a term containing A itself and other variables A'. A specialty of *S-CE* occurs if a variable A has to be constrained several times. In this case the individual constraints have to be combined to a union type in order to get a type that covers at least the values required by all the constraints.

Stage 2 (called *GIS* for generate idempotent substitution) takes the result of *S-CE* and transforms it into an idempotent substitution σ.[7] This is done by iteratedly inserting t_A for A in some $t_{A'}$ in order to eliminate A (and analogously the other restricted variables) from the substitution result. If t_A contains A itself, a recursive binding is introduced for a new variable X_A and A is replaced by X_A in t_A. A certain order of insertions and recursive bindings after a finite number of steps yields a substitution σ with no variable from $dom(\sigma)$ occurring in one of the substitution results.

The result of *CE* is a set Σ of substitutions fulfilling the following theorem:

Theorem 1 (correctness of *CE*). *Let t_1, t_2 be types. Let there exist a value $v \neq \bot$ such that*

$$\exists \rho \,.\, v \in \langle\!\langle \rho(t_1) \rangle\!\rangle \cap \langle\!\langle \rho(t_2) \rangle\!\rangle \,.$$

Then there exists a substitution $\sigma \in CE(t_1, t_2)$ such that

$$\exists \tau \,.\, v \in \langle\!\langle \tau \circ \sigma(t_1) \rangle\!\rangle \cap \langle\!\langle \tau \circ \sigma(t_2) \rangle\!\rangle \,.$$

Sketch of proof The proof consists of two steps: Step 1 shows that for every common element v of t_1 and t_2 there is a substitution σ' in the result of *S-CE* and a number $k_0 \in \mathbb{N}$ such that for every $k \geq k_0$:

$$\exists \tau \,.\, v \in \langle\!\langle \tau \circ {\sigma'}^k(t_1) \rangle\!\rangle \cap \langle\!\langle \tau \circ {\sigma'}^k(t_2) \rangle\!\rangle$$

where ${\sigma'}^k$ denotes applying σ' k times. This proof is done by structural induction on the types t_1 and t_2 and considering the different decomposition and constraining rules of *S-CE* independently.

[7] More precisely, *S-CE* returns several such constraint sets and each of them is transformed into an idempotent substitution independently.

Step 2 of the proof shows that for every substitution σ' *GIS* returns a substitution σ with

$$\sigma \circ \sigma'(t) = \sigma(t)$$

for every term t. With step 1 of the proof and induction on k_0 as chosen there this implies the theorem. $\square$

A definition of *CE* especially spotting on *S-CE* is given in [13]. [14] gives a full definition of *CE* and a detailed proof of Theorem 1.

The following examples illustrate the behaviour of *CE*:

Example 4. Let

$$t_1 = \mu X.(\cup\ \texttt{nil}\ (A\ .\ X)) \text{ and } t_2 = (\texttt{nat}\ .\ (\texttt{string}\ .\ \texttt{nil}))\,.$$

CE iteratedly unfolds the recursive type t_1 to the semantically equivalent type

$$(\cup\ \texttt{nil}\ (A\ .\ (\cup\ \texttt{nil}\ (A\ .\ (\cup\ \texttt{nil}\ (A\ .\ \mu X.(\cup\ \texttt{nil}\ (A\ .\ X)))))))))\,.$$

Checking this type with t_2 causes A to be instantiated with both *nat* and *string*. The result instantiation of A is a union type containing both of these instantiations:

$$CE(t_1, t_2) = \{\{A \leftarrow (\cup\ \texttt{string}\ \texttt{nat}\}\}\,.$$

Example 5. Consider the types t_1 and t_2 defined by

$$\begin{aligned} t_1 &:= (X\ .\ (Y\ .\ (Z\ .\ \texttt{nil}))) \\ t_2 &:= ((Y\ .\ X)\ .\ (Z\ .\ ((\texttt{num}\ .\ X)\ .\texttt{nil})))\,. \end{aligned}$$

The return value of *CE* is

$$\begin{aligned} CE(t_1, t_2) = \{\{ & X \leftarrow \mu V.(\mu W.(\texttt{num}\ .\ (W\ .\ V))\ .\ V), \\ & Y \leftarrow \mu W.(\texttt{num}\ .\ \mu V.(W\ .\ V)), \\ & Z \leftarrow \mu W.(\texttt{num}\ .\ \mu V.(W\ .\ V))\}\}\,. \end{aligned}$$

While Ex. 5 is a rather artificial example showing the generation of recursive binding for variables with mutually dependencies, Ex. 4 is quite important. It shows the correct detection of common elements of a list t_1 with variable element type and a list t_2 with elements of several different types.

4 Type Inference by Abstract Interpretation

The complete type inference process is presented in terms of an abstract interpretation. This follows [3] where abstract interpretation was shown to be appropriate to express different well known type inference systems.

To perform complete type inference, an abstract semantics for the purely functional subset of e.g. Scheme is straightforward for most program expressions (e.g. for generation and application of lambda-closures, for if-expressions, constants, ...). Environments are also abstracted in a natural manner.

The main difference to usual approaches used for sound type checkers is the processing of calls to predefined functions. Using the algorithm *CE* sketched in Sec. 3.2 the type inference system checks for every function call $(f\ a)$ with input type t_{in} of f[8] and type t_a inferred for a whether

$$\langle\!\langle t_a \rangle\!\rangle \cap \langle\!\langle t_{\text{in}} \rangle\!\rangle \neq \emptyset$$

holds. In the case of an empty intersection a type error is detected. Otherwise, the output substitutions of *CE* are used to restrict type variables. Special care has to be taken not to destroy the information which variables were restricted to certain types. This is necessary to enable further restrictions:

Example 6. Consider the following piece of code:

```
(if (null? l)
    1
    (+ (car l) 42))
```

The condition `(null? l)` introduces the following restrictions for l:

- $l \leftarrow$ `nil` for the *then*-case.
- $l \leftarrow \top \setminus$`nil` for the *else*-case. The expression `(car l)` further restricts l to a pair. This is just possible if l was not replaced by $\top \setminus$`nil` during the first restriction.

For recursive function definitions the problem of non-termination of the abstract interpretation occurs. It can be solved in two stages:

1. Whenever recursion is detected in a function definition, the system tries to find a fixpoint by an iterated process that is comparable to the approach described in [5].

[8] The input type is taken with respect to an I/O-representation of f, i.e. $dom(f) \subseteq t_{\text{in}}$ holds.

2. If the fixpoint iteration fails to reach a fixpoint then the computation is stopped after exceeding a threshold on the number of recursive calls. The intermediate types generated so far are generalized to a recursive type. If this generalization still is not a fixpoint then a further generalization to $\top$ is performed.[9]

With the abstract interpreter working as sketched before we can perform type inference for a program P as follows: For every user defined function f with arity k in P a call $(f\ A_1\ \dots\ A_k)$ with new type variables A_i is generated and interpreted abstractly. The I/O-representation denoting f is given by the restrictions of the A_i as input type and result of the abstract interpretation as output type.

A theorem stating the completeness of the resulting type checker is proven in [12]. Since due to lack of space this paper cannot present all the needed formal tools to formulate the mentioned theorem, we can just give an informal summary:

> Let e be an expression and e_A an abstraction of e, i.e. an abstract expression that is generated from e by replacing some of the constants occurring in e by corresponding types. Then abstractly interpreting e_A yields an abstract value v_A that contains the result v of interpreting e in the standard semantics.

The proof is done in several steps:

1. The statement is proven for expressions e not incorporating recursion. This is done inductively by case distinction on the structure of e.
2. Using a fixpoint argument the correctness of the recursion handling is proven. This is done without requiring termination.
3. The generalizations of types that are performed in the case of non-termination do not change the properties proven of the step before but enforce termination.

If the abstract interpretation of e_A yields the type `error` (just denoting type errors) as output result then by the statement sketched above the evaluation of e cannot succeed without a type error. The type checker is therefore complete in the sense that it accepts every program that can be executed without a runtime type error.

[9] Note that such generalizations cannot cause programs to be rejected as possible in sound type inference. However, we sometimes cannot avoid to overlook errors because of generalizations of types.

We finish the sketch of our complete type inference system by a well-typed and an ill-typed example of a program for multiplying two vectors given as lists of their elements.

Example 7 (well-typed example). Consider the following Scheme program for the multiplication of two vectors:

```
(define (v-v-mult row column)
  (cond ((and (null? row) (null? column)) 0)
         ;; non-recursive case
        (else
         (+ (* (car row) (car column))
                 ;; multiply first elements and process
                 ;; the vector rests recursively
            (v-v-mult (cdr row) (cdr column))))))
```

The result of type inference for this program is an I/O-presentation with a single input type

$$IN = \mu V_R.(\cup \; \texttt{nil} \; (\texttt{num} \; . \; V_R)) \quad \times \mu V_C.(\cup \; \texttt{nil} \; (\texttt{num} \; . \; V_C))$$

and the corresponding output type $OUT = \texttt{num}$.

Example 8 (ill-typed example). Now consider the following ill-typed modification of the program of Ex. 7:

```
(define (v-v-mult row column)
  (cond ((and (null? row) (null? column)) 0)
         ;; non-recursive case
        (else
         (+ (* (car row) (car column))
                 ;; multiply first elements and process
                 ;; the vector rests recursively
            (* row column))))) ; Ill-typed call to *
```

The result of type checking this program is a type error for the call to $*$ in the last line because *row* and *column* are lists which cannot be multiplied.

A detailed definition of the complete type inference system together with a step to step presentation of the examples above can be found in [12].

5 Conclusions

This paper motivated a new approach to type checking with the focus on completeness and sketched a complete type checker and a corresponding type inference system. By accepting every well-typed program and rejecting only provably ill-typed programs we can use more powerful type languages without restricting the set of accepted programs due to inaccuracies of the type checker and without enforcing explicit type annotations. E.g. subtyping as a step towards powerful type languages as found in e.g. [1], [8] or partial types as presented in [9, 10] can be processed by our type checker. Our type checker is described as an abstract interpretation and yields detailed support in detecting type errors. Furthermore, warnings of an additional sound soft typing system [2], [16] or a system with output similar to soft typing [4] can be used additionally to spot on those parts of the program that could not be proven to be well-typed.

A first approach to overcome the disadvantage of soft typing not to reject any programs is given in [15], but it depends strongly on a certain representation of a restricted type language. The work presented here is applicable to a wide range of type languages as e.g. a powerful one for Scheme (for which an example definition exists). It allows the building of powerful type checkers with subtyping that benefit from both strongly typed languages and soft typing.

References

1. 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.
2. R. Cartwright and M. Fagan. Soft typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.
3. P. Cousot. Types as abstract interpretations. In *Conference Record of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 316–331, Jan. 1997.
4. 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.
5. S. P. Jones and C. Clack. Finding fixpoints in abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis Horrwood, 1987.
6. R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.
7. R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, Dec. 1978.

8. G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, Dec. 1994.
9. S. R. Thatte. Type inference with partial types. *Springer Verlag (Heidelberg, FRG and NewYork NY, USA) LNCS 317, Automata, Languages and Programming, 15th International Colloquium*, pages 615–629, 1988.
10. S. R. Thatte. Quasi-static typing. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 367–381, San Francisco, California, Jan. 1990.
11. P. S. Wang. The undecidability of the existence of zeros of real elementary functions. *Journal of the ACM*, 21(4):586–589, Oct. 1974.
12. M. Widera. *Complete Type Inference in Functional Programming.* PhD thesis, University of Hagen, Germany, Apr. 2001. (submitted).
13. M. Widera and C. Beierle. Detecting common elements of types. In S. Gilmore, editor, *Trends in Functional Programming*, volume 2. Intellect, 2000.
14. M. Widera and C. Beierle. An approach to checking the non-disjointness of types in functional programming. Informatik Berichte 281, FernUniversität Hagen, Jan. 2001.
15. A. K. Wright. *Practical Soft Typing.* PhD thesis, Rice University, Houston, Texas, Aug. 1994.
16. 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.