Chapter 1

Function Types in Complete Type Inference

Manfred Widera[1] and Christoph Beierle[2]

Abstract: We study type checking that is complete in the sense that it accepts every program whose subexpressions can all be executed without raising a type error at runtime. In a complete type checker for every function call $(f\ a)$ of a function f with an argument expression a of type t_a it is checked whether f is applicable to one of the possible values of a, i.e. whether $\langle\![t_a]\!\rangle \cap dom(f) \neq \emptyset$ holds where $\langle\![t]\!\rangle$ denotes the semantics of a type t. When approximating $dom(f)$ by a type t_{in} it turns out that the usual function type constructor is not appropriate for complete type checking: for a function type $t_f = t_{\text{in}} \to t_{\text{out}}$ of f the input type t_{in} is usually not guaranteed to contain all values of $dom(f)$ and the test for common elements can erroneously fail. We therefore introduce an alternative notion of function types, called I/O-representation, where the input types cover a superset of the domain of the denoted functions. We show that this notion of function types fits into the framework of complete type checking much better than the usual function type constructor. Moreover, we argue that complete type checking overcomes the disadvantages of soft-typing approaches by enabling the *rejection* of programs instead of just raising a warning.

1.1 INTRODUCTION TO COMPLETE TYPE CHECKING

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.

[1]Fachbereich Informatik, FernUniversität Hagen, D-58084 Hagen, Germany. Email: `Manfred.Widera@Fernuni-Hagen.de`

[2]Fachbereich Informatik, FernUniversität Hagen, D-58084 Hagen, Germany. Email: `Christoph.Beierle@Fernuni-Hagen.de`

Soft-typing for dynamically typed languages (e.g. [CF91], [Fag92]), 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.

1.1.1 Disadvantages of Sound Type Checking

The usual type checkers are *sound* and are used either in strongly typed languages (e.g. Standard ML [Wik87] or Haskell [HPF97]) 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 [Mil78] "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:

Example 1.1. Consider the following function definition.[3]

```
(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 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 all possible inputs for y (e.g. [Wan74]). Thus, we cannot prove *with-div* free of type errors.[4]

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 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.

- For a dynamically typed language the results of a sound type checker are available via soft-typing systems, i.e. type systems that can suppress runtime type checks on provably well typed function calls and raise type warnings on other calls. Because of unprecise results of the type checker such a soft-typing system can raise 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.

[3] All example programs in this paper are written in Scheme.

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

A further problem for soft-typing is shown by the following example Ex. 1.2:

Example 1.2. 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 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 *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.

1.1.2 The Goal of Complete Type Checking

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.[5] *P goes wrong* if it 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.

Example 1.3. In the program of Ex. 1.2 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. 1.2) 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*

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

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.

Note that sound type checking of course detects all function calls that go wrong according to the definition above. However, as stated above, it also often detects function calls that do not go wrong. Soft-typing has the additional problem of not *rejecting* the calls that go wrong; e.g. soft-typing will rise just a warning even for $(* \ 'a \ 3)$, but it will not reject it. Soft-typing therefore does not force the programmer to react on the type checker's messages.

By completeness of a type checker we mean that a program *P* that does not 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. A definition of a complete type checker can be found in [WB00] and [Wid01].

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 a number of further 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.

1.1.3 Function Types and Complete Type Checking

The complete type checking process motivated so far is not easily implementable using the usual function type constructor. This is because complete type checking needs to know *all* valid input values of each function used in a checked program. This paper discusses the inappropriateness of the usual function type constructor for complete type checking and provides a solution be defining a new notion of function types.

So the aim of this paper is to clarify one part of the type language used for complete type checking (namely the function types) and therefore to provide a tool needed by complete type checkers with the desired properties.

The rest of the paper is organized as follows: in Sec. 1.2 the type language is sketched. After discussing the inappropriateness of the usual function type constructor in Sec. 1.3 the new notion of function types is introduced in Sec. 1.4 with the definition in Subsec. 1.4.1, a couple of examples in Subsec. 1.4.2, and a discussion of its influence on complete type inference in Subsec. 1.4.3. Section 1.5 gives some conclusions.

1.2 THE TYPE LANGUAGE

The type language that is used by our complete type checker 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 following notions of types:

- `posint` for positive integer numbers.
- `int` for integer numbers.
- `num` for arbitrary numbers.
- `string` for strings.
- `error` for type errors.
- `nil` for the type just denoting the empty list.
- $\top$ for the type of all values.
- $t_1 \times t_2$ or $(\times\ t_1\ \ldots\ t_n)$ for Cartesian products.
- $(\cup\ t_1\ \ldots\ t_n)$ for union types.
- $t_1 \setminus t_2$ for difference types denoting the values of t_1 that are not denoted by t_2.
- $(t_1\ .\ t_2)$ for cons pairs.
- $(\texttt{vector}\ t)$ for a vector type with argument type t.
- $\mu X.t$ for recursively defined types.
- $(\texttt{list}\ t)$ for lists of argument type t. This is an abbreviation for the recursive type definition $\mu X.(\cup\ \texttt{nil}\ (t\ .\ X))$.

The usual function types do not occur in the type language and will be replaced by I/O-representations in the following section.

The set of all types is denoted by T. For every type $t \in T$ the semantics of t, i.e. the set of values denoted by the type t, is denoted by $\langle\![t]\!\rangle$.

1.3 THE INAPPROPRIATENESS OF FUNCTION TYPES

1.3.1 Constraints used in Complete Type Checking

In a type language allowing subtyping the usual kind of constraints checked by a sound type checker is a set inclusion constraint. For a function call $(f\ a)$ and a type t_a inferred for a the system checks whether $\langle\![t_a]\!\rangle \subseteq dom(f)$ holds.

In the type system the set $dom(f)$ is approximated by the input type of f. If f has the type $t_{\text{in}} \rightarrow t_{\text{out}}$ then we can conclude $\langle\![t_{\text{in}}]\!\rangle \subseteq dom(f)$ and the usual check $\langle\![t_a]\!\rangle \subseteq \langle\![t_{\text{in}}]\!\rangle$ just succeeds if $\langle\![t_a]\!\rangle \subseteq dom(f)$ also succeeds.

Informally, the set inclusion constraints of sound type checkers describe the test whether *all* values denoted by the type t_a of the argument a can be processed by the function f. For complete type checking we want to test whether *any* of the values in the argument type can be processed. If this test fails then the considered function call must fail and the program can be rejected safely.

The test whether there are values in the argument type that cause an error free evaluation of the call can be expressed as follows: considering a call $(f\ a)$ as above the complete type checker performs the test $\langle\![t_a]\!\rangle \cap dom(f) \neq \emptyset$ and rejects the program if this test fail for any of the calls. Unfortunately, this test cannot be safely approximated by a test using t_{in} instead of $dom(f)$. We will discuss the problem and provide a solution in the rest of the paper.

1.3.2 Common Element Constraints and Function Types

The usual definition of function types (cf. e.g. [Mil78]) is done by a function type constructor $\rightarrow$ with the following semantics:
A type $t_1 \rightarrow t_2$ represents all functions f with

- $\langle\![t_1]\!\rangle \subseteq dom(f)$.
- $f(\langle\![t_1]\!\rangle) \subseteq \langle\![t_2]\!\rangle$ (where $f(S) = \{f(s) \mid s \in S\}$ for a set S).

This definition of function types is appropriate for sound type systems. For an argument type t and an input type t_1 those systems usually perform the test $\langle\![t]\!\rangle \subseteq \langle\![t_1]\!\rangle$ approximated by a subtype relation $\sqsubseteq$ with

$$t \sqsubseteq t_1 \Rightarrow \langle\![t]\!\rangle \subseteq \langle\![t_1]\!\rangle \subseteq dom(f)$$

This implies that the test $t \sqsubseteq t_1$ is a sound approximation of the test for the applicability of f to t.

In a complete type checker we rather want to approximate the test

$$\langle\![t]\!\rangle \cap dom(f) \neq \emptyset$$

i.e. instead of raising an error when *some* elements of t cause an error we raise an error when *all* elements of t are no valid arguments to f. Unfortunately, a function type defined in the way above is of no use for a complete type checker because there might be cases in which $\langle\![t]\!\rangle \cap dom(f) \neq \emptyset$, but $\langle\![t]\!\rangle \cap \langle\![t_1]\!\rangle = \emptyset$ raises a type error.

Example 1.4. According to the usual definition of function types the unary division operator should have the type $\texttt{num} \setminus \texttt{zero} \rightarrow \texttt{num}$. But because of the problem of deciding whether the argument is 0 one often states the division-by-zero problem to be outside the scope of the type checker.

1.4 THE NEW FUNCTION TYPE CONSTRUCTOR

Putting the division-by-zero problem into the scope of the type checker is easily possible for a complete type checker that can detect some of the division-by-zero

errors, but not all of them. To do this a complete type checker needs a function type definition different from that given above.

1.4.1 Definition of I/O-Representations

The complete type inference system/type checker we have in mind here uses the directed data flow properties of abstract interpretations to infer abstract predefined functions or abstract lambda closures for predefined and user defined functions, respectively, instead of function types. Since it seems quite difficult to find a short and understandable representation for the output of such function abstractions we will define I/O-representations for functions. The I/O-representations expressing the main properties of a function can be used for printing types of functions.

Definition 1.5 (I/O-representations). *An I/O-representation of a function is given by a set of I/O-representation pairs $IN_i \dashrightarrow OUT_i$ with $IN_i, OUT_i \in T$ such that:*

- $dom(f) \subseteq \bigcup_i \langle [IN_i] \rangle$
- $\forall_i f(\langle [IN_i] \rangle) \subseteq \langle [OUT_i] \rangle \cup \{\texttt{error}\}$

The set of all I/O-representations is denoted by `io`*.*

By the first part of the definition every value that can be processed by a function must be member of at least one IN_i. A (complete) type checker making use of I/O-representations cannot raise an error because of calling a function f with an input argument $v \in dom(f)$ that is not member of any of f's input types IN_i. Though I/O-representations look similar to refinement types [FP91] or types of functions constructed by using intersection types [Pie96] these notions of function types just guarantee to cover a subset of the domains of the denoted functions. In this property they differ from I/O-representations.
By the second part of the definition applying the function to an argument $v : IN_i$ cannot yield a result that is not of type OUT_i (except of error). Thus, uniting all those OUT_i with the corresponding IN_i having common elements with the argument type yields a type that covers all values possible for a function application.

Example 1.6 (I/O-representations of functions). Consider the following I/O-represention of the function *add* where every line represents one I/O-represention pair, e.g. one element of the I/O-representation set:

$$\begin{aligned} \texttt{posint} \times \texttt{posint} &\dashrightarrow \texttt{posint} \\ \texttt{int} \times \texttt{int} &\dashrightarrow \texttt{int} \\ \texttt{num} \times \texttt{num} &\dashrightarrow \texttt{num} \\ \texttt{string} \times \texttt{string} &\dashrightarrow \texttt{string} \end{aligned}$$

One could imagine that *add* is the usual addition on the three mentioned number types and the concatenation function on strings.

This example can furthermore illustrate the difference between I/O-representations and intersection types: by dropping the last line describing the transformation of strings by *add* the set of functions denoted by the I/O-representation gets smaller: functions processing strings are no longer denoted by this I/O-representation. However, the corresponding intersection type without information about strings not only still allows all functions transforming two strings into a string but also those functions not applicable to strings or returning output values of a different type. The set of denoted functions hence gets larger.

The set T of all types used for complete type inference is extended to contain I/O-representations. The notion of I/O-representations replaces the function type constructor used in usual type systems.

When a function f expects a function f' as input the inferred input type of f' is known just from inspecting the arguments of the calls to f'. Each argument type must have a non-empty intersection with the domain of f'. Thus, the valid input values of f' are not completely known, but we know types PIN_i each of which has common elements with the domain of f'. Furthermore, when an output type $POUT_i$ is inferred for a given input PIN_i we can just expect the property $f(\langle\![PIN_i]\!\rangle) \cup \langle\![POUT_i]\!\rangle \neq \emptyset$ to hold. Therefore, I/O-representations as defined before are to restrictive to express the information that can be inferred about higher order functions.

We solve this problem by defining the notion of a PI/PO-representation of f' for expressing the type of a function argument f' of a higher order function f. PI/PO-representations differ from I/O-representation especially in the meaning of the input types. While the input types of I/O-representations must be exhaustive in the sense that their union has to cover *all* values a denoted function is applicable to, the input types of PI/PO-representations only need to cover *some* of the values the denoted functions are applicable to.

Definition 1.7. (PI/PO-representations) *A PI/PO-representation of a function f is given by a set of PI/PO-representation pairs $PIN_i \dashrightarrow\!\!\!\!\!\approx \; POUT_i$ with types PIN_i and $POUT_i$ such that:*

- $\forall i \,.\, dom(f) \cap \langle\![PIN_i]\!\rangle \neq \emptyset$
- $\forall i \,.\, f(\langle\![PIN_i]\!\rangle) \cap \langle\![POUT_i]\!\rangle \neq \emptyset$

The set of all PI/PO-representations is denoted by `pipo`.

Example 1.8. **(PI/PO-representation)** Consider the following function *map1* implementing the usual map operation for just unary functions and one list and its use:

```
(define (map1 f l)
  (if (null? l) ()
      (cons (f (car l)) (map1 f (cdr l)))))

(define (usemap1 f l)
```

```
(let ((l-new (map1 f l)))
  (+ (car l-new) (car (cdr l-new)))))
```

An I/O-representation for *usemap1* is

$$\{A \overset{\approx}{\dashrightarrow} \texttt{num}\} \times (\mathit{list}\, A) \dashrightarrow (\mathit{list}\, \texttt{num})$$

From the use of *l-new* in *usemap1* we know that the output type of the first argument of *usemap1* must contain numbers. But as long as the input function is not known further output values might be possible. Analogously, from applying f to $(\mathit{car}\ l)$ we know that f must be applicable to some values of the element type of l. But again, f may be applicable to other values, too.

The definitions of I/O-representations and PI/PO-representations on the one hand provide a notion for printing function types. On the other hand they contain all necessary information used for type checking and type inference. For example, the needed properties of the abstractions of predefined function are completely given by their I/O-representations.

1.4.2 Examples

Since the use of PI/PO-representations is restricted to the input types of higher order functions, we will focus on I/O-representations in the following. In this subsection we will compare the usual function types and the I/O-representations of several functions that are predefined in many functional programming languages.

For those functions whose domains can be expressed exactly by the type language in use there is not much difference between usual function types and I/O-representations.

Example 1.9. Consider the functions *car* and *cdr* for selecting the first or second element of a pair, respectively. Their function types can be given as:

$$\mathit{car} : (A\ .\ \top) \to A$$
$$\mathit{cdr} : (\top\ .\ A) \to A$$

where A denotes a (implicitly universally quantified) type variable. Analogously, the I/O-representations of *car* and *cdr* are given as

$$\mathit{car} : \{(A\ .\ \top) \dashrightarrow A\}$$
$$\mathit{cdr} : \{(\top\ .\ A) \dashrightarrow A\}$$

Using both notions of function types we can express the types of *car* and `cdr` in a similar manner, because the types $(A\ .\ \top)$ and $(\top\ .\ A)$ of pairs with the first or second element of type A, respectively, can be expressed exactly in our type language.

One could imagine that the difference between usual function types and I/O-representations would become more obvious for functions whose domains must be *approximated* by the input types:

Example 1.10. Consider the division function $/$ and the function *vector-ref* for selecting a vector element with a given number. Using the usual function type constructor these functions have the following types:

$$/ : num \times num \to num$$
$$\textit{vector-ref} : (\texttt{vector}\ A) \times \texttt{int} \to A$$

Since usual function types and I/O-representations approximate the input types of the denoted functions from different sides (usual function types from below and I/O-representations from above) it is surprising that the I/O-representations of $/$ and *vector-ref* look similar to the usual function types, i.e.:

$$/ : \{num \times num \dashrightarrow num\}$$
$$\textit{vector-ref} : \{(\texttt{vector}\ A) \times \texttt{int} \dashrightarrow A\}$$

The reason for this similar appearance in contrast to different meanings of the constructors lies in the use of the usual function type constructor:

- Though $x \times 0 \notin dom(/)$ for every number x we have $x \times 0 \in \langle\!|\texttt{num} \times \texttt{num}|\!\rangle$.
- For a vector v of type $(\texttt{vector}\ A)$ with an arbitrary argument type A not all integer numbers are valid inputs to *vector-ref* in the second argument position. Though integers i that are negative or exceed the number of elements in v are invalid and yield argument tuples $v \times i \notin dom(\textit{vector-ref})$ these tuples are covered by the input type, i.e. $v \times i \in \langle\!|(\texttt{vector}\ A) \times \texttt{int}|\!\rangle$.

In both cases usual type systems use function types for the considered functions that cover values not contained in the domain of the respective function. The function type constructor $\to$ is used in a manner not respecting its semantics. To be precise, the function types given above are not function types for $/$ and *vector-ref*, respectively. This use of the function type constructor $\to$ makes it difficult to present the existing differences to the corresponding I/O-representations $\dashrightarrow$ (which are indeed correct I/O-representations of $/$ and *vector-ref*) in examples.

1.4.3 I/O-Representations and Complete Type Inference

As already stated, the motivation for defining the notion of I/O-representations is the inappropriateness of the usual function type constructor for complete type checking.

The use of I/O-representations is therefore given by the benefits of complete type checking, which can be illustrated by revisiting the motivating examples from Sec. 1.1:

Compared to the static type checker of a strictly typed language a complete type checker can work on a refined type language without raising errors for run-time correct calls:

Example 1.11. Reconsider the program of Ex. 1.1. By using complete type checking we can understand division by zero errors as type errors. In cases where the

second argument of the division operator can be proven to be zero, a type error is risen.

Compared to soft-typing a complete type checker can indeed reject a program in case of a *provable* type error.

Example 1.12. Reconsider the program defined in Ex. 1.2. For a complete type checker it is possible to infer the type $\{\texttt{num} \dashrightarrow \texttt{list num}\}$ for the function *generate*. ($\texttt{list}\ A$ is an abbreviation for a recursive definition of lists of type A.) Therefore, f expects a number as input and calls reverse with an argument of type $\texttt{list num}$. From this the system can infer the type $\texttt{num}$ for the result of the call $(car\ l)$ in line 6.

For the predefined function *append* we can assume the I/O-representation $\{\texttt{list}\ A \times \texttt{list}\ B \dashrightarrow \texttt{list}\ A \cup B\}$ to be provided.

Since the inferred type $\texttt{num}$ of line 6 and the expected type $\texttt{list}\ B$ of the second argument of *append* do not have common elements under any instantiation of B the type checker can detect a type error in the call to *append*.

Note that the type error in Ex. 1.12 leads to the detection of a piece of program code that under no circumstances can be executed without rising a runtime type error. Thus, a complete type checker can safely reject the program given in Ex. 1.2. The type checker developed in [Wid01] yields a rejection of this program by performing precisely the type inference steps indicated in the example above.

Unfortunately, it is impossible to give a detailed definition of complete type checking using I/O-representations in this paper due to lack of space. For such a definition see [Wid01].

1.5 CONCLUSION

Motivated by the requirements of complete type checking we have described the inappropriateness of the usual function type constructor. An alternative notion of function types called I/O-representation is introduced that fits better into the framework of complete type checking.

The new function types differ from the usual ones in the fact that they are not antimonotonic in the first argument and that their input types cover the whole domains of the denoted functions. For describing functions in an input argument of a higher order function PI/PO-representations yield an appropriate variant of I/O-representations with the input and output types not completely known.

Several examples showed the differences between usual and new function types. These differences are not obvious in many cases. As we have shown, the reason for this is an inappropriate use of the usual function type constructor by pushing certain misapplications of functions (e.g. division by zero or range errors of arrays/vectors) out of the scope of the type checker.

The new notion of I/O-representations can serve for several purposes including the external representation of types and storing the main properties of abstract predefined functions efficiently. A detailed discussion of the use of I/O-representations can be found in [Wid01].

Acknowledgments

We would like to thank the anonymous referees for their helpful comments.

REFERENCES

[CF91] Robert Cartwright and Mike Fagan. Soft-typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[Fag92] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, Houston, Texas, August 1992.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, Toronto, Ontario Canada, 26–28 June 1991. *SIGPLAN Notices* 26(6), June 1991.

[HPF97] Paul Hudak, John Peterson, and Joseph H. Fasel. *A Gentle Introduction to Haskell – Version 1.4 –*, March 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Pie96] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 11, 1996.

[Wan74] Paul S. Wang. The undecidability of the existence of zeros of real elementary functions. *Journal of the ACM*, 21(4):586–589, October 1974.

[WB00] Manfred Widera and Christoph Beierle. How to combine the benefits of strict and soft typing. In Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*. Intellect, 2000.

[Wid01] Manfred Widera. *Complete Type Inference in Functional Programming*. Mensch & Buch Verlag, Berlin, 2001. (PhD thesis, Dept. of Computer Science, FernUniversität Hagen).

[Wik87] Åke Wikström. *Functional Programming using Standard ML*. Prentice Hall, 1987.