Chapter 1

How to Combine the Benefits of Strict and Soft Typing

Manfred Widera[1] and Christoph Beierle[1]

Abstract: There is no abstract.

1.1 INTRODUCTION

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

Soft typing for dynamically typed languages (e.g. [CF91]), 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 chapter 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 checker supports powerful type languages including subtyping and is applicable to dynamically typed languages like Scheme.

The chapter is organised as follows: In Section 1.2 we give a motivation for complete type checking. Section 1.3 defines the main component of a complete type checker by abstract interpretation. Section 1.4 contains a summary of related work and some conclusions.

[1]Fachbereich Informatik, FernUniversität Hagen, D-58084 Hagen, Germany

1.2 THE USE OF COMPLETE TYPE CHECKING

Definition 1.1 (Dummy Definition). *This is just a pathetic dummy definition.*

1.2.1 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. That is, 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. Consider the function definition in Example 1.2.

Example 1.2. `(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. Because of this problem, the division by zero normally lies out of the scope of a sound type checker. □

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.
- 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 Example 1.3: that of being "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.

Example 1.3. Consider the following erroneous implementation of reverse and its use:

```
(define (reverse l)
  (if (null? l)
      '() ; reversed empty list is empty
    (append (reverse (cdr l))
              ; reverse rest
            (car l)))) ; should be (list (car l))
              ; 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. □

1.2.2 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 $a \notin dom(f)$.

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.[2] *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. Examples of calls conditionally going wrong are given in Example 1.4.

Example 1.4. In Example 1.3 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 Example 1.3) is $(f\ 3)$. Note that the call $(f\ 0)$ 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 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 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 a number of further advantages:

In debugging a program 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

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

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.

1.2.3 Realizing a Complete Type Checker

In powerful type languages 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 t be the type inferred for the argument of a function call, s 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 $t \sqsubseteq s$. 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 not by the exact type of the argument cause the test $t \sqsubseteq s$ to fail and the program to be rejected (cf Example 1.2 with $s = \texttt{num}$ and t 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 values of the exact type from those additionally in the inferred type the type checker rejects only calls that must go wrong for every value of the inferred type. Therefore, the complete type checker tests every call in the program for $\langle\!\![t]\!\!\rangle \cap \langle\!\![s]\!\!\rangle \neq \emptyset$ (where $\langle\!\![t]\!\!\rangle$ is the set of values of type t). Every call that does not fulfill this property is caused by an expression that conditionally goes wrong.

1.3 THE DEFINITION OF COMPLETE SUBTYPING

In this section we define the main component of a complete type checker for a functional language like Scheme or SML with dynamic typing. We use:

- a set *Sym* of symbols.
- lambda abstractions: $(lambda\ (x_1\ \ldots\ x_k)\ e)$
- function application: $(f\ e_1\ \ldots\ e_k)$
- tuple-like data constructors like *cons* for pairs.

The standard semantics is the usual one. The set of all values is denoted by *Value*. It contains as subsets a set *PFunc* of predefined functions and a set *LC* of lambda closures $lc(e(x_1,\ldots,x_k),E)$ where e is an expression, $x_1,\ldots,x_k \in Sym$ and E is an environment. The domain of predefined functions and lambda closures f is the set $dom(f) \subseteq Value$ of values f is applicable to without error. Environments $E \in Env$ can be understood as functions from symbols to values. The set of symbols bound in an environment (or abstract environment as given in Definition 1.10) E is denoted by $dom(E)$.

Definition 1.5 (type language). *Type expressions are given by:*

1. *A finite set* $\mathcal{B} = \{\bot, \top, b_1, \ldots, b_k\}$ *of type constants called base types.*

2. *A finite set* $\mathcal{C} = \{\cup, \cap, \mu, c_1, \ldots, c_n\}$ *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.* □

Due to lack of space we will omit the use of type variables in the following except of the typings of predefined functions.

Definition 1.6 (semantics of ground types). *Let* $t \in \mathcal{T}_G$ *be a ground type expression. t represents a set of values* $\langle\!\lbrack t \rbrack\!\rangle \subseteq$ *Value defined as follows:*[3]

- $\langle\!\lbrack \bot \rbrack\!\rangle := \{\bot\}$ *where* $\bot \in$ *Value expresses non-termination.*[4]
- $\langle\!\lbrack \top \rbrack\!\rangle :=$ *Value represents the set of all values.*
- *The base types* $b_1, \ldots, b_k$ *represent sets of simple values with the following property: If* $b, b' \in \mathcal{B}$, $\langle\!\lbrack b \rbrack\!\rangle \cap \langle\!\lbrack b' \rbrack\!\rangle \neq \emptyset$ *then there exists some* $\tilde{b} \in \mathcal{B}$ *with* $\langle\!\lbrack \tilde{b} \rbrack\!\rangle = \langle\!\lbrack b \rbrack\!\rangle \cap \langle\!\lbrack b' \rbrack\!\rangle$.
- $\langle\!\lbrack (\cup\ t_1\ \ldots\ t_k) \rbrack\!\rangle := (\cup\ \langle\!\lbrack t_1 \rbrack\!\rangle\ \ldots\ \langle\!\lbrack t_k \rbrack\!\rangle)$, $\langle\!\lbrack (\cap\ t_1\ \ldots\ t_k) \rbrack\!\rangle := (\cap\ \langle\!\lbrack t_1 \rbrack\!\rangle\ \ldots\ \langle\!\lbrack t_k \rbrack\!\rangle)$.
- $\mu X.t$ *with t containing X as free variable defines a recursive type as the least fixed point of the equation* $\langle\!\lbrack t' \rbrack\!\rangle = \langle\!\lbrack t[X \leftarrow t'] \rbrack\!\rangle$.[5]
- $c_1, \ldots, c_n$ *are free type constructors and correspond to tuple-like data constructors. E.g. for pairs* $\langle\!\lbrack (t_1\ .\ t_2) \rbrack\!\rangle := \{(v_1\ .\ v_2) \mid v_1 \in \langle\!\lbrack t_1 \rbrack\!\rangle, v_2 \in \langle\!\lbrack t_2 \rbrack\!\rangle\}$.

For $v \in \langle\!\lbrack t \rbrack\!\rangle$ *we also write* $v : t$. *Every type contains an additional value* `wrong` *denoting a type error. The type checker will detect an error for types not containing any value except* $\bot$ *and* `wrong`. □

Definition 1.7 (type hierarchy). *The subset relation* $\subseteq$ *on the power set of Value introduces a type hierarchy on the set* $\mathcal{T}_G$ *with the subtype relation* $\sqsubseteq_R$ *by the following definition:*

$$t_1 \sqsubseteq_R t_2 \iff \langle\!\lbrack t_1 \rbrack\!\rangle \subseteq \langle\!\lbrack t_2 \rbrack\!\rangle\,.$$

□

Though there are subtyping procedures for quite powerful type languages (e.g. [Dam94]) we use approximations of the type hierarchy of Definition 1.7. By this we do not rely on type languages for which an algorithm deciding this subtype relation exists.

[3] The semantics of types is denoted by $\langle\!\lbrack\cdot\rbrack\!\rangle$ instead of $\lbrack\!\lbrack\cdot\rbrack\!\rbrack$ because $\lbrack\!\lbrack\cdot\rbrack\!\rbrack$ will be used for the (abstract) semantics of programs.

[4] The value $\bot$ is also an element of every other type, but is usually omitted.

[5] $t[X \leftarrow t']$ is the type t with every free occurrence of the variable X replaced by t'.

Definition 1.8 (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* $t_1 \sqsubseteq t_2 \wedge t_2 \not\sqsubseteq t_1$*.* □

Our type language does not contain a function type constructor. The set of abstract values used in abstract interpretation contains functions defined as follows:

Definition 1.9 (abstract functions). *The set of abstract functions* $Func_A$ *consists of:*

- *Abstract predefined functions* $f \in PFunc_A$*.*
- *Abstract lambda-closures* $lc_A(e(x_1,\ldots,x_k),E) \in LC_A$ *where* e *is an expression,* $x_1,\ldots,x_k$ *are symbols and* $E \in Env_A$ *is an abstract environment as given in Definition 1.10.*

When just the I/O-Behaviour of an abstract function f *is of interest we express* f *by its I/O-representation* $D(f)$*, i.e. by a set of I/O-pairs* (IN_i, OUT_i) *where* IN_i, OUT_i *are types or further I/O-representations. The pairs* (IN_i, OUT_i) *are called* typings *of* f*.* □

The definition of abstract functions is unusual for type systems in distinguishing between predefined functions and lambda closures. As we will see later type errors are just detected in the calls to *predefined* functions. A construct quite similar to the usual function types is given by the I/O-representation of types. But their semantics as given in Definition 1.11 differs from the usual approach, too.

The set of abstract values is now given as $Value_A := \mathcal{T} \cup Func_A$.

Definition 1.10 (abstract environment). *An* abstract environment *is a function* $E : Sym \to Value_A$*. The set of abstract environments is denoted by* Env_A*.* □

Definition 1.11 (semantics of abstract functions). *Let* f *be an abstract function and* $r = D(f) =: \{(IN_1, OUT_1), \ldots, (IN_k, OUT_k)\}$ *the I/O-representation of* f*.*

- *If* $f \in PFunc_A$ *then* $\langle\![f]\!\rangle := std(f) \in PFunc$ *where* $std(f)$ *maps every abstract predefined function to a standard predefined function.*
- $f = lc_A(e(x_1,\ldots,x_k),E_A) \Rightarrow \langle\![f]\!\rangle := \{lc(e(x_1,\ldots,x_k),E) \in LC \mid E \in \langle\![E_A]\!\rangle\}$
- $\langle\![r]\!\rangle := \{g \in PFunc \cup LC \mid \exists i \in \{1,\ldots,k\} \,.\, dom(g) \cap \langle\![IN_i]\!\rangle \neq \emptyset, \forall i \,.\, g(dom(g) \cap \langle\![IN_i]\!\rangle) \subseteq \langle\![OUT_i]\!\rangle\}$ □

In contrast to usual function types $\langle\![r]\!\rangle$ contains all functions f whose domain *has common elements* with at least one input type and which transform these common elements to values of the corresponding output type.

Definition 1.12 (semantics of abstract environments). *The semantics of $E_A \in Env_A$ is $\langle\!\![E_A]\!\!\rangle := \{E \in Env \mid \forall x \in dom(E_A)\,.\,E(x) \in \langle\!\![E_A(x)]\!\!\rangle\}$.* $\square$

The idea of complete type checking applies in particular to calls of predefined functions. When $f \in PFunc_A$ is applied to an expression e where $[\![e]\!]$ (the abstract value of e) and no input type of f have common elements then a conditional type error (precisely the misapplied call 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.

Definition 1.13 (partial predefined application function). *Let $f \in PFunc_A$ be an abstract predefined function and let t' be an abstract value. The partial predefined application function $PPAF(f,t')$ is the set of all $s \in Value_A$ with:*

1. *$(t,s) \in \widehat{D(f)}$ where $\widehat{D(f)}$ is the set of ground instances of pairs in $D(f)$.*
2. *$CE(t,t')$ is* true *where CE is a predicate approximating the test for common elements with the property $\neg CE(t,t') \Rightarrow \langle\!\![t]\!\!\rangle \cap \langle\!\![t']\!\!\rangle = \emptyset$.*
3. *If $(\tilde{t},\tilde{s}) \in \widehat{D(f)}$ is a typing of f then the following holds:*
   (a) $t \cap t' \not\sqsubset \tilde{t} \cap t'$.
   (b) $t \cap t' = \tilde{t} \cap t' \Longrightarrow \tilde{t} \not\sqsubset t$. $\square$

Informally $PPAF(f,t')$ is the set of all $s \in Value_A$ that can occur as output from f (1) where the corresponding expected input t to f is the most special one (3b) that yields a maximal intersection with the provided input t' (3a). The test for common elements in (2) is the test of applicability of f to t'. The other conditions just enforce a result that is not more general than necessary.

Example 1.14. Consider an application $(+\ x\ y)$ and a set $D(+)$ of typings for $+$ with:

$$(\texttt{nat} \times \texttt{nat}, \texttt{nat}) \in D(+) \tag{1.1}$$
$$(\texttt{int} \times \texttt{int}, \texttt{int}) \in D(+) \tag{1.2}$$
$$(\texttt{num} \times \texttt{num}, \texttt{num}) \in D(+) \tag{1.3}$$
$$(\texttt{string} \times \texttt{string}, \texttt{string}) \in D(+) \tag{1.4}$$

(1.1), (1.2) and (1.3) denote the sum of naturals, integers and arbitrary numbers, respectively, with $\texttt{nat} \sqsubseteq \texttt{int} \sqsubseteq \texttt{num}$ and (1.4) denotes the concatenation of strings.

Using $A \times B$ or $(\times\ A_1\ \dots\ A_k)$ to denote a product type with 2 or k elements, if $x : \texttt{nat}$ and $y : \texttt{int}$ we calculate $PPAF(+, \texttt{nat} \times \texttt{int})$ as follows:

- $CE(t,t')$ is true for $t \in \{\texttt{nat} \times \texttt{nat}, \texttt{int} \times \texttt{int}, \texttt{num} \times \texttt{num}\}$.
- The intersection $t \cap t'$ is most general for $t \in \{\texttt{int} \times \texttt{int}, \texttt{num} \times \texttt{num}\}$.

TABLE 1.1. **Abstract semantics for a simple complete type checker**

$x \in Sym \Rightarrow$	$[\![x]\!](E) = E(x)$	(Sym)
	$[\![(lambda\ (x_1\ \ldots\ x_k)\ e)]\!](E) =$ $(lc_A(e(x_1,\ldots,x_k),E)$	(Lambda)
	$[\![(lc_A(e(x_1,\ldots,x_k),E')\ e_1\ \ldots\ e_k)]\!](E) =$ $[\![e]\!](E'[x_1 \mapsto [\![e_1]\!](E),\ldots,x_k \mapsto [\![e_k]\!](E)])$	(App-Lambda)
$f \in PFunc_A \Rightarrow$	$[\![(f\ e_1\ \ldots\ e_k)]\!](E) =$ $PAF(f,(\times\ [\![e_1]\!](E)\ \ldots\ [\![e_k]\!](E)))$	(App-Pre)

- Because of $\mathtt{int} \times \mathtt{int} \sqsubset \mathtt{num} \times \mathtt{num}$ we have $t := \mathtt{int} \times \mathtt{int}$.

As result we get $PPAF(+,\mathtt{nat} \times \mathtt{int}) = \{\mathtt{int}\}$.

If $x : (\cup\ \mathtt{nat}\ \mathtt{string})$ and $y : (\cup\ \mathtt{int}\ \mathtt{string})$ then (2) of Definition 1.13 is also fulfilled for $t = \mathtt{string} \times \mathtt{string}$ which fulfills (3a) and (3b), too. Thus, we get

$$PPAF(+,(\cup\ \mathtt{nat}\ \mathtt{string}) \times (\cup\ \mathtt{int}\ \mathtt{string})) = \{\mathtt{int},\mathtt{string}\}\,.$$

If $x : \mathtt{nat}$ and $y : \mathtt{string}$ then $t' := \mathtt{nat} \times \mathtt{string}$, but the definition of t fails since there is no input type t of $+$ with $CE(t,t')$. (Note that $t \times \bot = \bot \times t = \bot$.) □

As Example 1.14 shows, $PPAF(f,t')$ can have no, one or several elements. If it has no element a type error has been detected. In the case of several elements the function application has to be typed with the union of them:

Definition 1.15 (predefined application function). *Let f and t' be as in Definition 1.13. The* predefined application function *is defined as*

$$PAF(f,t') := \begin{cases} \bigcup_{\sigma \in PPAF(f,t')} \sigma & \text{if } PPAF(f,t') \neq \emptyset \\ \mathtt{error} & \text{else}\,. \end{cases}$$

□

With a compatible subtype relation $\sqsubseteq$ a simple complete (with respect to programs that conditionally go wrong) type checker is given by the abstract semantic function $[\![\cdot]\!](\cdot)$ presented in Table 1.1 where the first argument is an expression and the second argument is an abstract environment $E \in Env_A$.

A program is checked by starting the abstract interpretation of its main function with an abstract representation of the possible arguments as input and an abstract environment that binds the symbols used for predefined functions.

The new idea of completeness can be found in the semantic rule (*App-Pre*). It infers `error` for exactly those calls for which the inferred argument type contains no values the function is applicable to.

The type checker presented in Table 1.1 is a basic one. Further semantic rules e.g. for processing *let* or for *if* in a manner comparable to conditional types as presented in [AWL94] should be easy to include. A completeness result for this type checker is given in Theorem 1.16. An example of its work can be found in Example 1.17.

Theorem 1.16 (completeness of type checking). *Let CC be a type checker based on the abstract semantics presented in Table 1.1 with a compatible subtype relation $\sqsubseteq$. When CC only reports those calls $a = (f\ e)$ with $f \in PFunc$ with $[\![(f\ e)]\!](E) = \texttt{error}$ for the actual environment E then CC reports only calls in programs that conditionally go wrong.* □

Example 1.17. Consider the program from Example 1.3. Evaluating f causes the evaluation of *reverse* with the input $l \leftarrow (\mathit{list}\ \texttt{num})$, which in turn causes a message to be generated for *append*. (From the control flow information available during abstract interpretation one can identify the call $(\mathit{reverse}\ (\mathit{generate}\ n))$ in f as conditionally going wrong with the execution path to *append*. The condition for this path to be executed is $(\mathit{not}\ (\mathit{null?}\ (\mathit{generate}\ n)))$.) □

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 Example 1.3 the function f may never be called in a larger context.
- 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 context of Example 1.3 contains a call $(\mathit{reverse}\ '((1)\ (2)\ (3)))$ the expression $(\mathit{append}\ \ldots)$ will not generate a misapplication of *append* on this path.

As a result we cannot be sure whether a detected problem in the program will really cause a runtime 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). These can be used 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 for special cases like empty lists (like in Example 1.17). The motto behind this would be that one is not interested in functions that go wrong in *all* cases except for special cases like the empty list.

1.4 CONCLUSION AND FUTURE WORK

This chapter motivated a new approach to type checking with the focus on completeness and defined the main component of a complete type checker. 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. Subtyping as a step towards powerful type languages can be found in e.g. [AW93], [Smi94]. Our type checker is described as an abstract interpretation (as is justified by [Cou97]) and yields detailed support in detecting type errors. Further warnings produced by an additional sound soft typing system [CF91], [WC94], or a system with output similar to soft typing [FFK+96], can indicate those parts of the program that cannot be proven to be well-typed.

A first approach to overcome the disadvantage of soft typing not rejecting any programs is given in [Wri94], but it depends strongly on a restricted type language. The work presented here is applicable to a wide range of type languages and it allows the building of powerful type checkers with subtyping that benefit from both strongly typed languages and soft typing.

The formal definition of conditions that make a program *unacceptable* and the implementation of the corresponding checks are a subject of our current work.

REFERENCES

[AW93] A. S. Aiken and E. L. Wimmers. Type Inclusion Constraints and Type Inference. In *Conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 31–41, Copenhagen, Denmark, June 1993. ACM Press.

[AWL94] A. S. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *Symposium on Principles of Programming Languages (POPL'94)*, pages 163–173, Portland, Oregon, January 1994. ACM Press.

[CF91] R. Cartwright and M. Fagan. Soft Typing. In *Conference on Programming Language Design and Implementation (PLDI'91)*, pages 278–292, Toronto, Canada, June 1991.

[Cou97] P. Cousot. Types as Abstract Interpretations. In *Symposium on Principles of Programming Languages (POPL'97)*, pages 316–331, Paris, France, January 1997. ACM Press.

[Dam94] F. M. Damm. Subtyping with Union Types, Intersection Types and Recursive Types. In *International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, LNCS, Volume 789, pages 687–706, Sendai, Japan, April 1994. Springer-Verlag.

[FFK+96] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Conference on Programming Language Design and Implementation (PLDI'96)*, pages 23–32, Philadelphia, Pennsylvania, May 1996. ACM Press. SIGPLAN Notices 31(5).

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Smi94] G. S. Smith. Principal Type Schemes for Functional Programs with Overloading and Subtyping. *Science of Computer Programming*, 23(2–3):197–226, December 1994.

[Wan74] P. S. Wang. The Undecidability of the Existence of Zeros of Real Elementary Functions. *Journal of the ACM*, 21(4):586–589, October 1974.

[WC94] A. K. Wright and R. Cartwright. A Practical Soft Type System for Scheme. In *Conference on Lisp and Functional Programming (LFP'94)*, pages 250–262, Orlando, Florida, June 1994. ACM Press.

[Wri94] A. K. Wright. *Practical Soft Typing*. PhD thesis, Rice University, Houston, Texas, August 1994.