

A Type-based Framework for Automatic Debugging

Ozan Hafizoğulları Christoph Kreitz

Department of Computer Science, Cornell University
Ithaca, NY 14853, USA
{ozan,kreitz}@cs.cornell.edu

Abstract. We present a system for automatic debugging in typed functional languages. The system checks program properties specified by a user and finds bugs as well as conditions necessary to avoid them. It applies type-checking techniques built on top of the existing type system of the programming language. Its type system is based on the notion of *set types*, extended through type constructors, polymorphism and dependent types. Subtyping is used to allow finer specification. Type checking is achieved by collecting flow information through *flow types* and constraints on them. By solving the constraints we obtain instances for flow type variables and a set of conditions that make the typing provable. These conditions are converted into arithmetical formulae that can be checked by a common decision procedure. Our approach has a modular structure and can also be used for array bounds checking and other program analysis problems.

1 Introduction

Automatic debugging is the inference of program properties through automatic means with minimum user intervention. Systems for automatic debugging fill the area between two extremes. They are more ambitious than type checkers as they check more complex program properties. On the other hand, they have to perform their tasks fully automatically and cannot handle the same specifications as full-scale (interactive) program verifiers [32, 6]. There has been quite a sizeable amount of work in this area [7, 3, 39, 9, 10, 17] using different techniques for different programming paradigms, which accelerated with the growing emphasis on the array bounds checking problem due to the introduction of the *Java* language [19] and proof-carrying code [31].

The aim of our work is to provide a framework for automatic debugging, designed specifically for typed functional programming languages, that blends naturally with existing concepts from type theory and uses them effectively. An important aspect of this framework is conceptual clarity and a careful categorization of distinct parts of the analysis. This will lead to better insights into

¹ The authors are supported by ONR and DARPA under the respective contract numbers N00014-92-J-1764 and F30602-95-1-0047.

the problem, provide a conceptual basis for a comparison with corresponding systems for other (i.e. imperative, logical) programming paradigms, and make modifications for different problems easier. As a starting point for a prototypical automatic debugger we selected the ML type system [27, 25, 29] and its type inference algorithm.

Our system consists of three major components. The first is a *specification language*, which enables a user to communicate with the system and to make assertions about programs. In the context of imperative languages this language would be a decidable subset of Hoare-Dijkstra logic [12]. In the context of ML it would be the *type system*, in which coarse properties of the program, i.e. the type of some expression, can be asserted. Although there are other alternatives [5], we believe that type systems are the most natural way for this purpose in the context of higher order functional programming languages, as is evidenced by their proliferation. The type system that we developed for our framework is a refinement of the ML type system. In particular, we introduced a type $\{x \mid \varphi(x)\}$, which characterizes the collection of x that satisfy $\varphi(x)$ and is similar the set types of *Nuprl* [4] and predicate subtypes of *PVS* [6]. We also added type variables and polymorphism as well as subtyping capabilities using a variation of simple subtypes [28].

The other two components may be considered as part of *type checking*. One of them is the representation and collection of *flow information*. In this phase, whose imperative counter-part is data-flow analysis [1], equalities between types are collected and unified. We collect flow information through *flow type variables* and constraints on them. These constraints are solved through a process similar to the solution of data-flow equations, resulting in a substitution for the flow type variables and a set of variable-free constraints to be checked.

The last component represents the *constraints* to be checked in some relevant mathematical domain and checks them using a decision procedure. In this paper, we will convert constraints into arithmetical formulae and then use a common decision procedure [36] to check the validity of the latter.

The rest of the paper is organized as follows. In section 2, we introduce our type system for specifying and proving program properties. Type checking is covered in 3. Section 4 explains how to determine the validity of constraints. Section 5 discusses the consequences of adding arrays to the basic language. We discuss related work in section 6 and several design decisions, alternatives, and further applications of our method in section 7.

2 A Type System for Program Specification

The typed programming language that we consider in this paper is inspired by PCF [35] and corresponds to the core of ML. It contains operations on integers, boolean constants, conditionals, and (recursive) functions while the type system consists of integers, booleans, and the function type constructor.

$$\begin{aligned}
 e &:= i \mid \text{true} \mid \text{false} \mid x \mid \lambda x.e \mid e_1 e_2 \mid e_1 + e_2 \mid e_1 \leq e_2 \mid \text{if}(e_1, e_2, e_3) \mid \text{fix } f.e \\
 \tau &:= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2
 \end{aligned}$$

where i is any integer

Our type system is a refinement of the above core of the ML type system [27, 25, 29], which gives a coarse estimate of what kind of values an expressions e can take. For example, if e has type `int`, it is an integer, provided its evaluation terminates. Our *refinement type system* takes this one step further. We *refine* `int` and the other types by taking their subtypes into consideration. A type $\{x \mid \varphi(x)\}$ stands for the collection of integers x that satisfy the property $\varphi(x)$. The logic that a user may use to specify program properties is basically arithmetic extended by booleans, as described in Figure 1.

$$\begin{array}{l}
\text{Terms:} \quad t_{\text{int}} := x_{\text{int}} \mid i \mid t_{1\text{int}} + t_{2\text{int}} \\
\quad \quad \quad t_{\text{bool}} := x_{\text{bool}} \mid \text{true} \mid \text{false} \\
\\
\text{Assertions:} \quad \varphi := t_1 = t_2 \mid t_{1\text{int}} \leq t_{2\text{int}} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \neg \varphi
\end{array}$$

Fig. 1. User Logic

To deal with *polymorphism* we add refinement type variables s_{int} and s_{bool} . A type $b_1 \boxed{+} b_2$ is an approximation of $e_1 + e_2$, where b_1 and b_2 are respective approximations of e_1 and e_2 . Other primitive operations like subtraction, multiplication, case analysis etc. can be taken care of similarly in an implementation. $b_1 \sqcup b_2$, the *upper bound* of b_1 and b_2 , will be used, when the type checking algorithm concludes that both b_1 and b_2 happen to approximate a given expression in different parts of the computation.

The basic types are composed to form the types of the universe U_1 . U_2 -types are derived from U_1 -types to deal with polymorphism of refinement types. Constraint environments $\Pi s_1, \dots, s_n \mid C. \omega$ are added to be able to make finer assertions about the types.

Definition 1. The *refinement types* are defined as

Basic Types:

$$\begin{array}{l}
b_{\text{int}} := \{x \mid \varphi(x)\} \mid s_{\text{int}} \mid b_{1\text{int}} \boxed{+} b_{2\text{int}} \mid \boxed{\text{if}}(b_{1\text{bool}}, b_{2\text{int}}, b_{3\text{int}}) \mid b_{1\text{int}} \sqcup b_{2\text{int}} \\
b_{\text{bool}} := \{x \mid \varphi(x)\} \mid s_{\text{bool}} \mid b_{1\text{int}} \boxed{\leq} b_{2\text{int}} \mid \boxed{\text{if}}(b_{1\text{bool}}, b_{2\text{bool}}, b_{3\text{bool}}) \mid b_{1\text{bool}} \sqcup b_{2\text{bool}}
\end{array}$$

$$U_1\text{-types:} \quad r := b_{\text{int}} \mid b_{\text{bool}} \mid r_1 \rightarrow r_2 \mid \boxed{\text{if}}(b_{\text{bool}}, r_2, r_3)$$

$$U_2\text{-types:} \quad \omega := r \mid (\Pi s_1, \dots, s_n \mid C. \omega)$$

Note the similarity of the universes in refinement types and the corresponding mechanism in ML [25]. A semantics for this system based on ideals can be given using standard techniques [24].

Before presenting the type system, we have to give some definitions. We need a constraint environment to take care of subtyping relationships between the refinement types, very much in the spirit of [28]. The purpose is to be able to make finer typing assertions when we consider refinement type variables s_i .

Definition 2. *Type and constraint environments* are defined as

$$\begin{array}{l}
C_{\text{env}} := [] \mid (b_1 \sqsubseteq b_2) :: C_{\text{env}} \mid \neg(b_1 \sqsubseteq b_2) :: C_{\text{env}} \\
\Gamma := [] \mid (x : \omega) :: \Gamma
\end{array}$$

$$\begin{array}{c}
\frac{\vdash \varphi(i)}{\Gamma, C_{\text{env}} \triangleright i : \{x \mid \varphi(x)\}} \quad (\text{INT}) \\
\frac{\vdash \varphi(\text{true})}{\Gamma, C_{\text{env}} \triangleright \text{true} : \{x \mid \varphi(x)\}} \quad (\text{TRUE}) \\
\frac{\vdash \varphi(\text{false})}{\Gamma, C_{\text{env}} \triangleright \text{false} : \{x \mid \varphi(x)\}} \quad (\text{FALSE}) \\
\frac{\Gamma, x : r_1, C_{\text{env}} \triangleright e : r_2}{\Gamma, C_{\text{env}} \triangleright \lambda x. e : r_1 \rightarrow r_2} \quad (\text{LAM}) \\
\frac{}{\Gamma, x : r, C_{\text{env}} \triangleright x : r} \quad (\text{VAR}) \\
\frac{\Gamma, C_{\text{env}} @ C \triangleright e : r}{(\text{free_var}(C_{\text{env}}) \cup \text{free_var}(\Gamma)) \cap \{s_1, \dots, s_n\} = \emptyset} \quad (\text{II-INTRO}) \\
\frac{\Gamma, C_{\text{env}} \triangleright e : (\Pi s_1, \dots, s_n \mid C. r)}{C_{\text{env}} \vdash C[b_1, \dots, b_n / s_1, \dots, s_n]} \quad (\text{II-ELIM}) \\
\frac{}{\Gamma, C_{\text{env}} \triangleright + : (\Pi s_1, s_2 \mid . s_1 \rightarrow s_2 \rightarrow (s_1 \boxed{+} s_2))} \quad (\text{PLUS}) \\
\frac{}{\Gamma, C_{\text{env}} \triangleright \leq : (\Pi s_1, s_2 \mid . s_1 \rightarrow s_2 \rightarrow (s_1 \boxed{\leq} s_2))} \quad (\text{LEQ}) \\
\frac{\Gamma, C_{\text{env}} \triangleright e_1 : b_1}{\Gamma, C_{\text{env}}, \{\text{true}\} \sqsubseteq b_1 \triangleright e_2 : r_2} \\
\frac{\Gamma, C_{\text{env}}, \{\text{false}\} \sqsubseteq b_1 \triangleright e_3 : r_3}{\Gamma, C_{\text{env}} \triangleright \text{if}(e_1, e_2, e_3) : \boxed{\text{if}}(b_1, r_2, r_3)} \quad (\text{IF}) \\
\frac{\Gamma, C_{\text{env}} \triangleright e_1 : r_1 \rightarrow r_2}{\Gamma, C_{\text{env}} \triangleright e_2 : r_1} \quad (\text{APP}) \\
\frac{\Gamma, f : \omega, C_{\text{env}} \triangleright e : \omega}{\Gamma, C_{\text{env}} \triangleright \text{fix } f. e : \omega} \quad (\text{FIX})
\end{array}$$

Fig. 2. Type System for Refinement Types

The type system is given in Figure 2. The rules INT, TRUE and FALSE basically state that the constant c in question has the basic refinement type $\{x \mid \varphi(x)\}$ if it satisfies the condition $\varphi(c)$. The rule PLUS follows the computational meaning of the primitive function $+$: if s_1 and s_2 approximate the values of e_1 and e_2 then the value of $e_1 + e_2$ is approximated by $s_1 \boxed{+} s_2$. The rule LEQ is similar, while the rules LAM, APP, VAR and FIX are standard typing rules.

The rule IF can be read as follows. Assume e_1 is approximated by b_1 . If true is in b_1 then for some part of the execution, the left branch of the *if*-statement is taken. This is expressed by the second assumption, where e_2 is type-checked under the assumptions $\Gamma, C_{\text{env}}, \text{and}\{\text{true}\} \sqsubseteq b_1$. The situation is similar for false. As a consequence $\text{if}(e_1, e_2, e_3)$ is given the type $\boxed{\text{if}}(b_1, r_2, r_3)$, which is a finer approximation than the upper bound $r_2 \sqcup r_3$ chosen by most other methods.

The II-INTRO and II-ELIM rules correspond to those in ML. In II-INTRO, we have to respect an eigenvariable condition on the constraint environments C_{env} and C . In II-ELIM, before instantiating the quantified type variables in $(\Pi s_1, \dots, s_n \mid C. r)$ with the concrete refinement types b_1, \dots, b_n , we need to check that these satisfy the constraints in C .

Our type system also contains a subtyping facility, presented in Figure 3. It follows the simple subtypes paradigm in [28]. The rules SUB and ARROW are standard. IF-SUB basically makes explicit the intuitive meaning of $\boxed{\text{if}}(b, r_2, r_3)$, where r_2 and r_3 are higher order.

The function Tlogic in rule BASIC converts the subtyping constraints into formulae in arithmetic with unary predicates. As an example, consider the proof

$$\begin{array}{c}
\frac{\Gamma, C_{\text{env}} \triangleright e : r_1}{C_{\text{env}} \vdash r_1 \sqsubseteq r_2} \text{ (SUB)} \qquad \frac{\vdash_{\text{arith}} \text{Tlogic}(C_{\text{env}} \vdash b_1 \sqsubseteq b_2)}{C_{\text{env}} \vdash b_1 \sqsubseteq b_2} \text{ (BASIC)} \\
\\
\frac{C_{\text{env}} \vdash b'_1 \sqsubseteq b_1, \quad b_2 \sqsubseteq b'_2}{C_{\text{env}} \vdash b_1 \rightarrow b_2 \sqsubseteq b'_1 \rightarrow b'_2} \text{ (ARROW)} \\
\\
\frac{}{\boxed{\text{if}}(b, r_2 \rightarrow r'_2, r_3 \rightarrow r'_3) = \boxed{\text{if}}(b, r_2, r_3) \rightarrow \boxed{\text{if}}(b, r'_2, r'_3)} \text{ (IF-SUB)}
\end{array}$$

where

$$\begin{aligned}
\text{Tlogic}(C_{\text{env}} \vdash b_1 \sqsubseteq b_2) &= \text{let } [\varphi_1; \dots; \varphi_k] = [\varphi | \rho' \epsilon C_{\text{env}} \wedge \varphi = \text{Tlogic1}(\rho')] \text{ and } \varphi_0 = \text{Tlogic1}(b_1 \sqsubseteq b_2) \\
&\quad \text{in } \varphi_1 \wedge \dots \wedge \varphi_k \rightarrow \varphi_0 \\
\text{Tlogic1}(b_1 \sqsubseteq b_2) &= \text{let } \{x \mid \varphi_1(x)\} = \text{Tset}(b_1) \text{ and } \{y \mid \varphi_2(y)\} = \text{Tset}(b_2) \\
&\quad \text{in } \forall x. \varphi_1(x) \rightarrow \varphi_2(x) \\
\text{Tlogic1}(\neg b_1 \sqsubseteq b_2) &= \text{let } \{x \mid \varphi_1(x)\} = \text{Tset}(b_1) \text{ and } \{y \mid \varphi_2(y)\} = \text{Tset}(b_2) \\
&\quad \text{in } \neg \forall x. \varphi_1(x) \rightarrow \varphi_2(x) \\
\text{Tset}(\{x \mid \varphi(x)\}) &= \{x \mid \varphi(x)\} \\
\text{Tset}(s_i) &= \{x \mid P_i(x)\} \\
\text{Tset}(b_1 \boxed{+} b_2) &= \text{let } \{x \mid \varphi_1(x)\} = \text{Tset}(b_1) \text{ and } \{y \mid \varphi_2(y)\} = \text{Tset}(b_2) \\
&\quad \text{in } \{z \mid \exists x, y. \varphi_1(x) \wedge \varphi_2(y) \wedge z = x + y\} \\
\text{Tset}(b_1 \boxed{\leq} b_2) &= \text{let } \{x \mid \varphi_1(x)\} = \text{Tset}(b_1) \text{ and } \{y \mid \varphi_2(y)\} = \text{Tset}(b_2) \\
&\quad \text{in } \{z \mid \exists x, y. \varphi_1(x) \wedge \varphi_2(y) \wedge (z = \text{true} \leftrightarrow x \leq y) \wedge (z = \text{false} \leftrightarrow \neg(x \leq y))\} \\
\text{Tset}(\boxed{\text{if}}(b_1, b_2, b_3)) &= \text{let } \{x \mid \varphi_1(x)\} = \text{Tset}(b_1) \text{ and } \{y \mid \varphi_2(y)\} = \text{Tset}(b_2) \text{ and } \{z \mid \varphi_3(z)\} = \text{Tset}(b_3) \\
&\quad \text{in } \{w \mid \exists x, y, z. \varphi_1(x) \wedge \varphi_2(y) \wedge (z = \text{true} \leftrightarrow x \leq y) \wedge (z = \text{false} \leftrightarrow \neg(x \leq y))\} \\
\text{Tset}(b_1 \sqcup b_2) &= \text{let } \{x \mid \varphi_1(x)\} = \text{Tset}(b_1) \text{ and } \{y \mid \varphi_2(y)\} = \text{Tset}(b_2) \\
&\quad \text{in } \{x \mid \varphi_1(x) \vee \varphi_2(x)\}
\end{aligned}$$

Fig. 3. Subtyping Rules

of $\emptyset \vdash \{x \mid 0 \leq x \leq 5\} \sqsubseteq \{x \mid 0 \leq x \leq 7\}$. The function **Tlogic** converts this into $\forall x. (0 \leq x \leq 5) \rightarrow (0 \leq x \leq 7)$, which is provable in arithmetic. **Tlogic** works as follows. Each constraint is converted by **TlogicOne** to arithmetic in a manner similar to the translation of the subset relation into logic. **TlogicOne** uses the function **Tset** to get a set-like representation of the types in consideration.

Tset functions mostly in an intuitive fashion. The type $\{x \mid \varphi(x)\}$ is transformed into itself. The refinement type variable s_i is transformed to $\{x \mid P_i(x)\}$, where P_i is a predicate variable corresponding to s_i , since in a typing s_i can be replaced by any basic type, in particular $\{x \mid \varphi(x)\}$. The handling of $b_1 \boxed{+} b_2$ is obvious, while $b_1 \boxed{\leq} b_2$ follows the same idea. Here, the operation $z = (x \leq y)$, where z is a boolean, is transformed into arithmetic as $(z = \text{true} \leftrightarrow x \leq y) \wedge (z = \text{false} \leftrightarrow \neg(x \leq y))$. $\boxed{\text{if}}(b_1, b_2, b_3)$ and $b_1 \sqcup b_2$ are handled similarly.

3 Type Checking

We now introduce a type checking algorithm for our type system. The algorithm consists of two phases: flow-information gathering through constraints and a decision procedure for arithmetic. We first give an example to illustrate the ideas.

Example 1. Assume that we want to check the specification $e : r = \lambda x, y. x + y : s_1 \rightarrow s_2 \rightarrow \{x \mid 0 \leq x \leq 7\}$ in the environment $C_{\text{env}} = [s_1 \sqsubseteq \{x \mid 0 \leq x \leq 2\}, s_2 \sqsubseteq \{x \mid 0 \leq x \leq 3\}]$. To do so, we execute the function $\text{Debug}(\Gamma, C_{\text{env}} \triangleright e : r)$, where the type environment Γ is initially \emptyset .

Debug first *annotates* (or *decorates*) the typing $\Gamma \triangleright e$ with *flow type variables* α_i . As a result we get $\Gamma_\alpha \triangleright e^\rho = \text{annotate}(\Gamma \triangleright e)$ with $\Gamma_\alpha = \emptyset$ and $e^\rho = (\lambda x^{\alpha_1}, y^{\alpha_2}. (x^{\alpha_3} +^{\alpha_4 \rightarrow \alpha_5 \rightarrow \alpha_6} y^{\alpha_7})^{\alpha_8})^{\alpha_9 \rightarrow \alpha_{10} \rightarrow \alpha_{11}}$. We then compute a set $C_1 := \text{CC}(\Gamma_\alpha \triangleright e^\rho)$ of constraints between these variables that describe flow information: $C_1 = [\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_8 \sqsubseteq \alpha_9 \rightarrow \alpha_{10} \rightarrow \alpha_{11}, \alpha_1 \sqsubseteq \alpha_3, \alpha_2 \sqsubseteq \alpha_7, \alpha_3 \sqsubseteq \alpha_4, \alpha_7 \sqsubseteq \alpha_5, \alpha_6 \sqsubseteq \alpha_8, (\alpha_4 \boxplus \alpha_5) \sqsubseteq \alpha_6]$.¹ We also compute the constraints coming from the type environment Γ and the assumed type r : $C_2 := \text{CC}_{\text{env}}(\Gamma_\alpha \triangleright e^\rho : r) = [\alpha_9 \rightarrow \alpha_{10} \rightarrow \alpha_{11} \sqsubseteq s_1 \rightarrow s_2 \rightarrow \{x \mid 0 \leq x \leq 7\}]$. These are the constraints the user wants the system to check.

In the following step we extract a new set of constraints that is equivalent to $C_1 @ C_2$ but contains no function types. This step corresponds to the decomposition of higher order types in the unification phase of ML. We get $C := \text{decompose}(C_1 @ C_2) = [\alpha_9 \sqsubseteq \alpha_1, \alpha_{10} \sqsubseteq \alpha_2, \alpha_8 \sqsubseteq \alpha_{11}, \alpha_1 \sqsubseteq \alpha_3, \alpha_2 \sqsubseteq \alpha_7, \alpha_3 \sqsubseteq \alpha_4, \alpha_7 \sqsubseteq \alpha_5, \alpha_6 \sqsubseteq \alpha_8, s_1 \sqsubseteq \alpha_9, s_2 \sqsubseteq \alpha_{10}, \alpha_{11} \sqsubseteq \{x \mid 0 \leq x \leq 7\}]$. Next, we *pump* the refinement types through the flow type variables in order to determine the values that α_j should take to satisfy the constraints. As result we get a pair $\langle C', i \rangle := \text{pump}(C)$ where $i = [\alpha_1, \alpha_3, \alpha_4, \alpha_9 := s_1; \alpha_2, \alpha_5, \alpha_7, \alpha_{10} := s_2; \alpha_6, \alpha_8, \alpha_{11} := (s_1 \boxplus s_2)]$ is a substitution that provides the types each α_i would take in a proof of the typing and $C' = [(s_1 \boxplus s_2) \sqsubseteq \{x \mid 0 \leq x \leq 7\}]$ is a set of constraints without flow variables that must be satisfied for $i(C)$ to be valid. To determine the substitution i we iteratively search for flow variables α_j with the property that in all constraints $R \sqsubseteq \alpha_j$ the type R is a “constant”, and extend i such that $i(\alpha_j) = \sqcup \{R \mid R \sqsubseteq \alpha_j \text{ in } C\}$. This step corresponds to the unification of type variables in ML.

Finally, we check whether the assumptions $C_{\text{env}} = [s_1 \sqsubseteq \{x \mid 0 \leq x \leq 2\}, s_2 \sqsubseteq \{x \mid 0 \leq x \leq 3\}]$ imply the constraints $C' = [(s_1 \boxplus s_2) \sqsubseteq \{x \mid 0 \leq x \leq 7\}]$. As this is the case, we can conclude $\vdash \Gamma, C_{\text{env}} \triangleright e : r$. This last step corresponds to the checking of equalities between constant types in ML.

Before presenting our type checking algorithm in detail (Figures 4 and 5), we introduce a new construct $\langle \Gamma_1, C_1 \triangleright e : \omega \rangle$. This construct, whose intended meaning is $\vdash \Gamma_1, C_1 \triangleright e : \omega$, will allow the user to annotate subexpressions of a program. During the analysis of a program, the type checking algorithm will check whether $\vdash \Gamma_1, C_1 \triangleright e : \omega$ is true and if so, use it in checking the program further. As the algorithm will find the best estimates for non-recursive expressions, we will assume all the annotated subexpressions to be of the form $\text{fix } f.e'$. The construct corresponds to the annotation of loop invariants in imperative programs [12]. We will further discuss this construct in section 7.

¹ Note the similarity with the setting of data-flow equations in [1] or with the collection of type equalities in the type inference of ML. There are variations of ML type inference, but it can be devised so that type equalities are collected first in a distinct phase and unified in a second phase.

$$\text{Debug}(\Gamma, C_{\text{env}} \triangleright e : r) = \text{let } \Gamma_\alpha \triangleright e^\rho = \text{annotate}(\Gamma \triangleright e) \\ \text{in} \\ \text{TI}(\Gamma_\alpha, C_{\text{env}} \triangleright e^\rho : r)$$

$$\text{annotate}(\Gamma \triangleright e) = \text{let } \Gamma_\alpha = \mathbf{a}_3(\Gamma) \text{ and } e^\rho = \mathbf{a}_2(e) \text{ in } \Gamma_\alpha \triangleright e^\rho$$

where

$$\begin{aligned} \mathbf{a}_1(\text{int}) &= \alpha_1 & \mathbf{a}_1(\text{bool}) &= \alpha_2 & (\alpha_1, \alpha_2 \text{ new}) \\ \mathbf{a}_1(\tau_1 \rightarrow \tau_2) &= \mathbf{a}_1(\tau_1) \rightarrow \mathbf{a}_1(\tau_2) \\ \mathbf{a}_2(i) &= i^{\alpha_1} \\ \mathbf{a}_2(\text{true}) &= \text{true}^{\alpha_2} & \mathbf{a}_2(\text{false}) &= \text{false}^{\alpha_3} & (\alpha_1, \alpha_2, \alpha_3 \text{ new}) \\ \mathbf{a}_2(+) &= +^{\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3} & \mathbf{a}_2(\leq) &= \leq^{\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3} & (\alpha_i \text{ new}) \\ \mathbf{a}_2(x^\tau) &= x^{\mathbf{a}_1(\tau)} & \mathbf{a}_2((\lambda x^{\tau_1}. e^{\tau_2})^{\tau_1 \rightarrow \tau_2}) &= (\lambda x^{\mathbf{a}_1(\tau_1)}. \mathbf{a}_2(e^{\tau_2}))^{\mathbf{a}_1(\tau_1) \rightarrow \mathbf{a}_1(\tau_2)} \\ \mathbf{a}_2((e_1^{\tau_1} e_2^{\tau_2})^{\tau_3}) &= (\mathbf{a}_2(e_1^{\tau_1}) \mathbf{a}_2(e_2^{\tau_2}))^{\mathbf{a}_1(\tau_3)} \\ \mathbf{a}_2(\text{if}(e_1^{\text{bool}}, e_2^{\tau_2}, e_3^{\tau_2})) &= \text{if}(\mathbf{a}_2(e_1), \mathbf{a}_2(e_2), \mathbf{a}_2(e_3))^{\mathbf{a}_1(\tau_2)} \\ \mathbf{a}_2(\langle \Gamma_1, C_1 \triangleright (\text{fix } f.e)_\tau : \omega \rangle) &= \langle \mathbf{a}_3(\Gamma_1), C_1 \triangleright (\text{fix } f.\mathbf{a}_2(e)) : \omega \rangle^{\mathbf{a}_1(\tau)} \\ & \quad (\tau \text{ is the ordinary type of } (\text{fix } f.e)) \end{aligned}$$

$$\begin{aligned} \mathbf{a}_3([\]) &= [\] & \mathbf{a}_3((x_\tau : r) :: \Gamma) &= (x^{\mathbf{a}_1(\tau)} : r) :: (\mathbf{a}_3(\Gamma)) \\ \mathbf{a}_3((x : \Pi s_1, \dots, s_n | C.r) :: \Gamma) &= (x : \Pi s_1, \dots, s_n | C.r) :: (\mathbf{a}_3(\Gamma)) \end{aligned}$$

$$\begin{aligned} \text{TI}(\Gamma_\alpha, C_{\text{env}} \triangleright e^\rho : r) &= \text{if } \text{check_assertions}(\Gamma_\alpha, C_{\text{env}} \triangleright e^\rho) \\ & \text{then} \\ & \quad \text{let } C_1 = \text{CC}(\Gamma_\alpha \triangleright e^\rho) \\ & \quad \text{and } C_2 = \text{CC}_{\text{env}}(\Gamma_\alpha \triangleright e^\rho : r) \\ & \quad \text{and } \langle C', i \rangle = \text{pump}(\text{decompose}(C_1 @ C_2)) \\ & \quad \text{in} \\ & \quad \quad \text{checkB}(C_{\text{env}} \vdash C') \\ & \text{else} \\ & \quad \text{false} \end{aligned}$$

$$\begin{aligned} \text{check_assertions}(\Gamma_\alpha, C_{\text{env}} \triangleright \langle \Gamma_1, C_1 \triangleright (\text{fix } f.e)^\rho : (\Pi s_1, \dots, s_n | C.r) \rangle^{\rho'}) \\ = \text{TI}(\Gamma_1, f : (\Pi s_1, \dots, s_n | C.r), C_1 @ C[s'_1, \dots, s'_n / s_1, \dots, s_n] \triangleright \\ e^\rho : r[s'_1, \dots, s'_n / s_1, \dots, s_n]) \quad (s'_1, \dots, s'_n \text{ new}) \end{aligned}$$

$$\begin{aligned} \text{pump}(C) &= \text{letref } C' := C \text{ and } i := \text{id} & (\text{id is the identity substitution}) \\ & \text{for all } \alpha \text{ with 'for all } (R \sqsubseteq \alpha) \epsilon C', R \text{ basic type} \\ & \text{do} \\ & \quad \text{lower} := \sqcup \{b \mid b \sqsubseteq \alpha \epsilon C'\}; \\ & \quad i := [\alpha := \text{lower}] \circ i; \\ & \quad C' := \{C' - \{(b \sqsubseteq \alpha) \epsilon C'\}\}[\text{lower}/\alpha]; \\ & \text{od} \\ & i := ([\alpha := \{x \mid \text{true}\} \mid \alpha \in \text{free_var}(C')]) \circ i \\ & \text{in} \\ & \quad \langle C', i \rangle \end{aligned}$$

continued in Figure 5

Fig. 4. Type Checking Algorithm for Refinement Types I

$$\begin{aligned}
\text{decompose}([\] &= [\] \\
\text{decompose}([R_1 \sqsubseteq R_2]) &= [R_1 \sqsubseteq R_2], && \text{(if } R_1, R_2 \text{ are atomic)} \\
\text{decompose}([R_1 \rightarrow R_2 \sqsubseteq R'_1 \rightarrow R'_2]) &= [R'_1 \sqsubseteq R_1; R_2 \sqsubseteq R'_2] \\
\text{decompose}([\text{if}](R_1, R_2 \rightarrow R'_2, R_3 \rightarrow R'_3) \sqsubseteq R) & \\
= \text{decompose}([\text{if}](R_1, R_2, R_3) \rightarrow [\text{if}](R_1, R'_2, R'_3) \sqsubseteq R) & \text{(similarly for the other direction)} \\
\text{decompose}(C_1 @ C_2) &= \text{decompose}(C_1) @ \text{decompose}(C_2) \\
\text{decompose}(\text{IF}(b, C_2, C_3)) &= \text{IF}(b, \text{decompose}(C_2), \text{decompose}(C_3))
\end{aligned}$$

$$\begin{aligned}
\text{checkB}(C_{\text{env}} \vdash [\]) &= \text{true} \\
\text{checkB}(C_{\text{env}} \vdash [b_1 \sqsubseteq b_2]) &= \text{checkArith}(\text{c2logic}(C_{\text{env}} \vdash b_1 \sqsubseteq b_2)) \\
\text{checkB}(C_{\text{env}} \vdash C_1 @ C_2) &= \text{checkB}(C_{\text{env}} \vdash C_1) \text{ and } \text{checkB}(C_{\text{env}} \vdash C_2) \\
\text{checkB}(C_{\text{env}} \vdash \text{IF}(b, C_2, C_3)) & \\
= \text{checkB}(C_{\text{env}}, \{\text{true}\} \sqsubseteq b \vdash C_2) \text{ and } \text{checkB}(C_{\text{env}}, \{\text{false}\} \sqsubseteq b \vdash C_3)
\end{aligned}$$

$$\begin{aligned}
\text{CC}(\Gamma_\alpha \triangleright i^\alpha) &= [\{i\} \sqsubseteq \alpha] && \text{(similarly for true and false)} \\
\text{CC}(\Gamma_\alpha \triangleright +^{\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3}) &= [(\alpha_1 \boxed{+} \alpha_2) \sqsubseteq \alpha_3] && \text{(similarly for } \leq) \\
\text{CC}(\Gamma_\alpha \triangleright x^\rho) &= \text{if } (x^{\rho'} : r) \in \Gamma_\alpha \\
&\quad \text{then } [\rho' \sqsubseteq \rho] \\
&\quad \text{else if } (x : \Pi s_1, \dots, s_n \mid C.r) \in \Gamma_\alpha \\
&\quad \quad \text{then } (C[\alpha_1, \dots, \alpha_n / s_1, \dots, s_n] && (\alpha_1, \dots, \alpha_n \text{ new}) \\
&\quad \quad \quad @ [r[\alpha_1, \dots, \alpha_n / s_1, \dots, s_n] \sqsubseteq \rho]), \\
\text{CC}(\Gamma_\alpha \triangleright (\lambda x^{\rho_1}. e^{\rho_2})^{\rho_3 \rightarrow \rho_4}) &= \text{CC}(\Gamma_\alpha, x^{\rho_1} \triangleright e^{\rho_2}) @ [\rho_1 \rightarrow \rho_2 \sqsubseteq \rho_3 \rightarrow \rho_4] \\
\text{CC}(\Gamma_\alpha \triangleright (e_1^{\rho_1 \rightarrow \rho_2} e_2^{\rho_3})^{\rho_4}) &= \text{CC}(\Gamma_\alpha \triangleright e_1^{\rho_1 \rightarrow \rho_2}) @ \text{CC}(\Gamma_\alpha \triangleright e_2^{\rho_3}) @ [\rho_3 \sqsubseteq \rho_1; \rho_2 \sqsubseteq \rho_4] \\
\text{CC}(\Gamma_\alpha \triangleright [\text{if}](e_1^{\rho_1}, e_2^{\rho_2}, e_3^{\rho_3})^\rho) &= \text{let } C_1 = \text{CC}(\Gamma_\alpha \triangleright e_1^{\rho_1}) \\
&\quad \text{and } C_2 = \text{CC}(\Gamma_\alpha \triangleright e_2^{\rho_2}) \\
&\quad \text{and } C_3 = \text{CC}(\Gamma_\alpha \triangleright e_3^{\rho_3}) \\
&\quad \text{in} \\
&\quad C_1 @ \text{IF}(\rho_1, C_2, C_3) @ [\text{if}](\rho_1, \rho_2, \rho_3) \sqsubseteq \rho]
\end{aligned}$$

$$\begin{aligned}
\text{CC}(\Gamma_\alpha \triangleright \langle \Gamma_1, C_1 \triangleright e : (\Pi s_1, \dots, s_n \mid C.r) \rangle^\rho) &= \\
\text{mk_constraints}(\Gamma_\alpha \sqsubseteq (\Gamma_1 \uparrow) [\beta_1, \dots, \beta_n / s'_1, \dots, s'_m]) & \\
@ C_1 [\beta_1, \dots, \beta_n / s'_1, \dots, s'_m] & \\
@ C [\alpha_1, \dots, \alpha_n / s_1, \dots, s_n] [\beta_1, \dots, \beta_n / s'_1, \dots, s'_m] & \\
@ [r[\alpha_1, \dots, \alpha_n / s_1, \dots, s_n] [\beta_1, \dots, \beta_n / s'_1, \dots, s'_m] \sqsubseteq \rho] &
\end{aligned}$$

where $\{s'_1, \dots, s'_m\} = \text{free_var}(\Gamma_1) \cup \text{free_var}(C_1) \cup \text{free_var}(r)$, $(\Gamma_1 \uparrow) = [x^r \mid (x^\rho : r) \in \Gamma_1]$, $\alpha_i \beta_j$ new

$$\begin{aligned}
\text{mk_constraints}(\Gamma_1 \sqsubseteq \Gamma_2) &= [\rho_1 \sqsubseteq \rho_2 \mid x^{\rho_1} \in \Gamma_1 \text{ and } x^{\rho_2} \in \Gamma_2], \\
&\quad (x : (\Pi s_1, \dots, s_n \mid C.r)) \in \Gamma_2 \text{ for all } (x : (\Pi s_1, \dots, s_n \mid C.r)) \in \Gamma_1.
\end{aligned}$$

$$\begin{aligned}
\text{CC1}_{\text{env}}(\Gamma_\alpha \triangleright e^\rho : r) &= \text{CC1}_{\text{env}}(\Gamma_\alpha) @ [\rho \sqsubseteq r] \\
\text{CC1}_{\text{env}}([\]) &= [\] \\
\text{CC1}_{\text{env}}((x^\rho : r) :: \Gamma_\alpha) &= [r \sqsubseteq \rho] @ \text{CC1}_{\text{env}}(\Gamma_\alpha) \\
\text{CC1}_{\text{env}}((x : (\Pi s_1, \dots, s_n \mid C'.r)) :: \Gamma_\alpha) &= \text{CC1}_{\text{env}}(\Gamma_\alpha)
\end{aligned}$$

Fig. 5. Type Checking Algorithm for Refinement Types II

Our algorithm, which is presented in a top-down fashion in Figures 4 and 5, is based on the concept of *flow types*. This concept is defined as follows.

Definition 3. A *flow type* R is defined like ordinary refinement types r , with the addition of flow type variables α_i to basic refinement types b_{int} and b_{bool} .

The main program is the function `Debug`, which checks whether a given typing $\Gamma, C_{\text{env}} \triangleright e : r$ is provable. It consists of two function calls. `annotate`($\Gamma \triangleright e$) decorates the program with flow variables α_i while `TI`($\Gamma_\alpha, C_{\text{env}} \triangleright e^\rho : r$) takes a decorated typing and decides whether it is provable in the type system.

The function `TI` has to perform several tasks. As a user can annotate the program with assumptions like $\vdash \Gamma_1, C_1 \triangleright e : \omega$, it first has to check the correctness of these annotations by calling a function `checkAssertions`. Then it has to collect the constraints that give the flow information of the program. The function `CC`, which we developed for this purpose, has some similarity with the setting of data-flow equations in [1]. However, the constraints collected by `CC` are no longer represented by a list but are structured like a tree, which enables us to reason precisely about the semantics of `if`. The satisfaction relation on constraints defined below makes the need for structure explicit.

Definition 4.

1. The system C of *constraints* is defined by:

$$C := [] \mid [R_1 \sqsubseteq R_2] \mid C_1 @ C_2 \mid \text{IF}(b_{\text{bool}}, C_2, C_3)$$

2. The *satisfaction relation* \vdash on C is defined by the following rules:

$$\frac{}{C_{\text{env}} \vdash []} \qquad \frac{C_{\text{env}} \vdash R_1 \sqsubseteq R_2}{C_{\text{env}} \vdash [R_1 \sqsubseteq R_2]}$$

$$\frac{C_{\text{env}} \vdash C_1 \quad C_{\text{env}} \vdash C_2}{C_{\text{env}} \vdash C_1 @ C_2} \qquad \frac{C_{\text{env}} @ [\{\text{true}\} \sqsubseteq R] \vdash C_2 \quad C_{\text{env}} @ [\{\text{false}\} \sqsubseteq R] \vdash C_3}{C_{\text{env}} \vdash \text{IF}(R, C_2, C_3)}$$

Note how the fourth rule for the satisfaction relation \vdash follows the typing rule `IF`. The constraints in the environment C_{env} , however, are still ordinary lists.

The definition of `CC`($\Gamma_\alpha \triangleright e^\rho$) essentially follows the corresponding typing rules in Figure 2. The cases for the constants i , `true` and `false` are straightforward: if $\vdash \Gamma, C_{\text{env}} \triangleright i^{S(\alpha)}$ for some substitution S , we expect $\{i\} \sqsubseteq S(\alpha)$ to hold. The case for `+` is just an instantiation of the rule `PLUS`. From the conclusion of that rule, we can derive $\vdash \Gamma, C_{\text{env}} \triangleright + : S(\alpha_1) \rightarrow S(\alpha_2) \rightarrow S(\alpha_1) \boxed{+} S(\alpha_2)$, which is reflected in the constraint `CC` produces. Similarly, the cases for λ -abstractions and applications are straightforward. The case for a single variable x follows the rules `VAR` and `SUB`, and the case for `if`(e_1, e_2, e_3) follows `IF`. In the case for $\langle \Gamma_1, C_1 \triangleright e : (II s_1, \dots, s_n \mid C.r) \rangle^\rho$ we must make use of the assumption $\vdash \Gamma_1, C_1 \triangleright e : (II s_1, \dots, s_n \mid C.r)$, which is assumed to be proven by `checkAssertions` previously. Constraints are provided to make sure that the type of e in the current context is an instance of $(II s_1, \dots, s_n \mid C.r)$. Further constraints make sure that the assumptions in instantiated Γ_1 follow from the global type environment Γ_α and C_{env} implies the instantiated C_1 .

In TI the constraints determined by CC are joined with those arising from the type environment, using the function CC_{env} . We then apply the function decompose to get rid of higher order (i.e. function) types and then split the result C , using $\text{pump}(C)$, into a set C' of constraints without flow variables and a substitution i for the flow variables α_i such that $i(C)$ is valid if C' is. This corresponds to the solution of data-flow equations in [1].

The functions defined up to this point are fairly general and can be used for other program analysis problems as well, as we will discuss in section 7. Checking the validity of the constraints C' , however, is a domain specific problem. We have defined a function checkB for this purpose that relies on a conversion function c2logic that converts a constraint $C_{\text{env}} \vdash C$ into an arithmetical formula φ , and a decision procedure checkArith for arithmetical formula with unary predicates. c2logic is equivalent to the function Tlogic defined in Figure 3. Suitable arithmetical decision procedures will be discussed in section 4.

In the following we will give a series of lemmata and theorems that describe the properties of the functions in our algorithm and eventually show that our scheme works as intended. For this purpose we introduce a technical definition.

Definition 5.

1. $\Gamma_\alpha, e^\rho \triangleright r$ is *annotate-correct* if all the refinement types appearing in $\Gamma_\alpha, e^\rho \triangleright r$ have the same shape as the corresponding ordinary ML types of the expression up to the isomorphism defined by the subtyping rule IF-SUB.
2. $R_1 \sqsubseteq R_2$ is a *matching constraint* if the shapes of R_1 and R_2 are the same up to the isomorphism of defined by the subtyping rule IF-SUB.

In the rest of this section we will assume $\Gamma_\alpha \triangleright e^\rho$ to be annotate-correct. It can easily be shown that in this case the all the constraints that we are dealing with will be matching, and hence we will not mention this explicitly. Under this assumption we can show that all the functions used by TI behave as intended.

Lemma 1.

1. If $\text{check_assertions}(\Gamma_\alpha, C_{\text{env}} \triangleright e^\rho) = \text{true}$ then $\vdash \langle \Gamma_1, C_1 \triangleright e_1 : \omega_1 \rangle$ is in e^ρ for all $\langle \Gamma_1, C_1 \triangleright e_1 : \omega_1 \rangle$ in e^ρ .
2. Let $C_1 = \text{CC}(\Gamma_\alpha \triangleright e^\rho)$ and assume $C_{\text{env}} \vdash i(C_1)$. Then, $\vdash [x : i(\rho') \mid (x^{\rho'} : r) \in \Gamma_\alpha], C_{\text{env}} \triangleright e : i(\rho)$.
3. Let $C_2 = \text{CC}_{\text{env}}(\Gamma_\alpha \triangleright e^\rho : r)$. Assume $\vdash [x : i(\rho') \mid (x^{\rho'} : r) \in \Gamma_\alpha], C_{\text{env}} \triangleright e : i(\rho)$ and $C_{\text{env}} \vdash i(C_2)$. Then $\vdash \Gamma_\alpha, C_{\text{env}} \triangleright e : r$.
4. Let $C_2 = \text{decompose}(C_1)$. Then C_2 is decomposed, i.e. does not contain higher order type, and $C_2 \vdash C_1$ and $C_1 \vdash C_2$, i.e. $C_1 \equiv C_2$.
5. If C is decomposed, $\langle C', i \rangle = \text{pump}(C)$ and $C_{\text{env}} \vdash C'$, then $C_{\text{env}} \vdash i(C)$.
6. If $\text{checkB}(C_{\text{env}} \vdash C) = \text{true}$, then $C_{\text{env}} \vdash C$.

The correctness of checkAssertions , CC , and decompose is proven by structural induction. The proofs for CC_{env} and checkB are straightforward applications of the definitions. The correctness of the function pump is shown by induction on the number of iterations of its loop.

The correctness of the function `TI` follows now from its definition and lemma 1.

Lemma 2. *If $\text{TI}(\Gamma_\alpha, C_{\text{env}} \triangleright e^\rho : r) = \text{true}$, then $\vdash \Gamma, C_{\text{env}} \triangleright e : r$, where Γ is generated from Γ_α by erasing the decoration with flow types.*

As a consequence of lemma 2 we get that the typing $\Gamma, C_{\text{env}} \triangleright e : r$ is provable in the type system if $\text{Debug}(\Gamma, C_{\text{env}} \triangleright e : r) = \text{true}$. Thus the automatic debugging scheme explained in this section works as intended.

Theorem 1. *If $\text{Debug}(\Gamma, C_{\text{env}} \triangleright e : r) = \text{true}$, then $\vdash \Gamma, C_{\text{env}} \triangleright e : r$.*

4 Checking the Validity of Constraints

In this section we briefly discuss decision problem for logical formulae resulting from the translation of the refinement type constraints into arithmetic-logical formulae. In general, the decision problem for arithmetic is undecidable. Even the decision problem for Presburger Arithmetic has doubly exponential lower bound [8]. For this reason, modern arithmetical proof procedures decide Presburger arithmetical formulae with quantifiers at the outermost level only, all of which being universal [36]. We will do the same in our system.

Our logic is arithmetic with booleans and unary predicates. We can easily convert a formula with booleans to one without them, using standard techniques. For the elimination of unary predicates we use a technique called *hoisting*, which we explain in the example below.

Example 2. Consider $C_{\text{env}} = [s_1 \sqsubseteq \{x \mid 0 \leq x \leq 2\}, s_2 \sqsubseteq \{x \mid 0 \leq x \leq 3\}]$ and $C' = (s_1 \boxplus s_2) \sqsubseteq \{x \mid 0 \leq x \leq 7\}$ be the constraint environment and the constraints to be checked from Example 1. In order to check $C_{\text{env}} \vdash C$ we first have to compute $\varphi = \text{c2logic}(C_{\text{env}} \vdash C)$, such that $\vdash \varphi$ implies $C_{\text{env}} \vdash C'$. We get $\varphi = [(\forall x. P_1(x) \rightarrow (\exists y. 0 \leq y \leq 2 \wedge y = x)) \wedge (\forall z. P_2(x) \rightarrow (\exists w. 0 \leq w \leq 3 \wedge z = w))] \rightarrow (\forall x, y. P_1(x) \wedge P_2(y) \rightarrow (\exists z. 0 \leq z \leq 7 \wedge z = x + y))$.

In the next step we convert φ into a normal form $\psi = \forall x_1, \dots, x_n. \psi_1$ for some ψ_1 without quantifiers or unary predicates. Under certain assumptions we can generate a formula ψ' by selecting $P_i(t) := (t = X_i)$ for each P_i , where the X_i are new variables, then universally quantifying X_i at the outermost level, so that $\vdash \psi$ implies $\vdash \varphi$. We call this process *hoisting*, because the variables in question are hoisted to the top level. In this case we get $\psi' = \forall X_1, X_2. [(\forall x. x = X_1 \rightarrow (\exists y. 0 \leq y \leq 2 \wedge y = x)) \wedge (\forall z. z = X_2 \rightarrow (\exists w. 0 \leq w \leq 3 \wedge z = w))] \rightarrow \forall x, y. x = X_1 \wedge y = X_2 \rightarrow (\exists z. 0 \leq z \leq 7 \wedge z = x + y)$. Now ψ' is equivalent to $\psi = \forall X_1, X_2. (0 \leq X_1 \leq 2) \wedge (0 \leq X_2 \leq 3) \rightarrow (0 \leq X_1 + X_2 \leq 7)$, which can be proven by some common decision procedure for arithmetic [36].

One point that deserves noting is that *hoisting* becomes unnecessary if we restrict the basic types $\{x \mid \varphi(x)\}$ to intervals $[a, b] = \{x \mid a \leq x \leq b\}$. In this scheme ($[a, b] \sqsubseteq [c, d]$) is translated to $c \leq a \leq b \leq d$. The type variables can themselves be regarded as $[\delta, \gamma]$ where δ and γ are variables. ($[a, b] \boxplus [c, d]$) can be modelled as $[a + c, b + d]$ and the same idea can be used for other type constructors. The usage of intervals is especially attractive if our main concern is array bound checking, as discussed in section 5.

5 Extension to Arrays

An important application of our scheme is array bounds checking, which got renewed interest with the introduction of the language *Java* [19] and proof carrying code [31]. We model an array of k elements of type r simply as a function from $\{x \mid 0 \leq x \leq (k-1)\}$ to r . Array indexing is modelled as function application. The typing rules for arrays are as follows:

$$\frac{\Gamma, C_{\text{env}} \triangleright e_0 : r, \dots, e_k : r}{\Gamma, C_{\text{env}} \triangleright \{e_0, \dots, e_k\} : \{x \mid 0 \leq x \leq k\} \rightarrow r}$$

$$\frac{\Gamma, C_{\text{env}} \triangleright e_1 : b \rightarrow r \quad \Gamma, C_{\text{env}} \triangleright e_2 : b \quad \Gamma, C_{\text{env}} \triangleright e_3 : r}{\Gamma, C_{\text{env}} \triangleright e_1.(e_2) : r} \quad \frac{\Gamma, C_{\text{env}} \triangleright e_1 : b \rightarrow r \quad \Gamma, C_{\text{env}} \triangleright e_2 : b \quad \Gamma, C_{\text{env}} \triangleright e_3 : r}{\Gamma, C_{\text{env}} \triangleright e_1.(e_2).:=e_3 : \text{unit}}$$

The extension of the function CC follows directly from these typing rules.

$$\begin{aligned} \text{CC}(\Gamma_\alpha \triangleright \{e_0^{\rho_0}; \dots; e_k^{\rho_k}\}^{\alpha \rightarrow \rho}) &= \text{CC}(\Gamma_\alpha \triangleright e_0^{\rho_0}) @ \dots @ \text{CC}(\Gamma_\alpha \triangleright e_k^{\rho_k}) \\ &\quad @ [\rho_0 \sqsubseteq \rho; \dots; \rho_k \sqsubseteq \rho] @ [\alpha \sqsubseteq \{x \mid 0 \leq x \leq k\}] \\ \text{CC}(\Gamma_\alpha \triangleright e_1^{\rho_1 \rightarrow \rho_2} . (e_2^{\rho_3}) \rho_4) &= \text{let } C_1 = \text{CC}(\Gamma_\alpha \triangleright e_1^{\rho_1 \rightarrow \rho_2}) \\ &\quad \text{and } C_2 = \text{CC}(\Gamma_\alpha \triangleright e_2^{\rho_3}) \\ &\quad \text{in} \\ &\quad C_1 @ C_2 @ [\rho_3 \sqsubseteq \rho_1; \rho_2 \sqsubseteq \rho_4] \\ \text{CC}(\Gamma_\alpha \triangleright (e_1^{\rho_1 \rightarrow \rho_2} . (e_2^{\rho_3}) := e_3^{\rho_4})^{\rho_5}) &= \text{let } C_1 = \text{CC}(\Gamma_\alpha \triangleright e_1^{\rho_1 \rightarrow \rho_2}) \\ &\quad \text{and } C_2 = \text{CC}(\Gamma_\alpha \triangleright e_2^{\rho_3}) \\ &\quad \text{and } C_3 = \text{CC}(\Gamma_\alpha \triangleright e_3^{\rho_4}) \\ &\quad \text{in} \\ &\quad C_1 @ C_2 @ C_3 @ [\rho_3 \sqsubseteq \rho_1; \rho_4 \sqsubseteq \rho_2; \{x \mid \text{true}\} \sqsubseteq \rho_5] \end{aligned}$$

6 Related Work

In [39] Xi and Pfenning report about a system for checking ML programs with user annotations. Like Hayashi in his system ATTT [15] they base their ideas on singleton types and dependent types, although they are not explicit about the type system or how the flow information is generated. As the singleton types do not directly follow that of ML, their system is likely to be more complicated and it is not clear to what extent it can be generalized to other analysis problems.

In his thesis [10], Freeman introduces another notion of refinement for ML types. His types are defined as a finite lattice for each basic type. Finiteness is crucial for the type inference algorithm to converge. The sample lattices are small and only very simple properties like the emptiness of a list can be checked. Dealing with arithmetic, specification of operations on types, constraints on type variables, and a precise approximation for conditional statements is not possible and the collection of flow information is not mentioned. On the other hand, he uses intersection and union types. We expect that an extension of our system by intersection types at the U_1 level will not create substantial problems, but this

will not add enough observable increase in the quality of the analysis to justify the increased conceptual complexity.

Set constraints in different forms [17, 16, 2] are a valuable tool for program analysis. Aiken and Wimmers [2] give a type system using set constraints in which every pure lambda term has a type. For us this was not an option, as we want to check programs written in an existing typed functional programming language like ML. On the other hand [17, 16] does not provide a specification language for communication between the user and the system. Arithmetic is handled in a very rough fashion and it is not clear how decision procedures like SupInf [36] can be incooperated into the algorithm. We believe that solving constraints through flow variables and the function `pump` is conceptually simpler than the approach in [2, 16]. Also, since the semantics in [17] differs from that commonly used for ML [24], our system is conceptually more natural and does not have problems to cooperate with a general logical programming environment like PVS or Nuprl [32, 6].

A system built using set constraints as in [16] is *MrSpidey* [9]. Its emphasis, however, is on soft typing for Scheme and in that sense the properties it checks roughly correspond to the ML type system. In particular, complex logical assertions on the input program are not possible.

The *ESC* system of Digital Systems Research Center [7] and *Syntox* by Bourdoncle [3] are two examples of automatic debugging for imperative languages. ESC uses a variation of Floyd-Hoare Logic [12] for user annotations. In the available documentation, it is not clear how the flow information is collected. But *verification conditions* (logical assertions) are generated and then checked by decision procedures. Syntox, on the other hand, is based on abstract interpretation by intervals $[a, b]$.

7 Discussion

We have presented a type-based system for automatic debugging in typed functional languages. Our type system is based on the notions of simple types, set types, subtyping, type constructors, polymorphism and dependent types, while type checking is achieved by collecting flow information through flow types and constraints on them.

An extension of our system by the let-construct *let* $x = e_1$ *in* e_2 to the language can be accomplished by duplicating the constraints for e_1 for each occurrence of x in e_2 and adding the constraints for flow into and out of that occurrence of x . Naturally, this scheme can be improved by caching the information from the previous instantiations.

In the previous sections, we used the construct $\langle I_1, C_1 \triangleright e : \omega \rangle$ to allow a user to make assertions about recursive functions. This construct corresponds to the user annotation of loop invariants in imperative languages [12]. Because of the modularity of our approach, we could also integrate techniques for *synthesizing* these invariants, but one should keep in mind that the synthesis of loop invariants is generally undecidable and that automated methods [16, 26, 13, 37]

are very limited. On the other hand, by using regular tree grammars [11] and defining $\text{CC}(\Gamma_\alpha \triangleright (\text{fix } f^{\rho_1}.e^{\rho_2})^{\rho_3}) := [\rho_2 \leq \rho_1 \leq \rho_3]@\text{CC}(\Gamma, f^{\rho_1} \triangleright e^{\rho_2})$ as in [14], our framework can also be used without the $\langle \Gamma_1, C_1 \triangleright e : \omega \rangle$ construct.

We have introduced our automated debugging system as a type *checker*, which means that result of $\text{Debug}(\Gamma, C_{\text{env}} \triangleright e : r)$ can only be `true` or `false`, assuming that the constraint environment C_{env} and the type r has been provided by the user. A simple variation of our scheme would be to have the function Debug *infer* both C_{env} and r such that $\vdash \Gamma, C_{\text{env}} \triangleright e : r$ holds. In this setting the input types of r would be assigned distinct refinement type variables s_i , the output types would be assigned $\{x \mid \text{true}\}$ (no information), and the goal would be to find constraints C_{env} on the input types s_i that ensure that the expression e respects a set of general well-definedness conditions. An example of such a condition would be that array bounds are never violated.

It is important to note that there is a trade-off between the efficiency of the debugging process and the quality of the analysis. In the previous sections we have assumed that the function `annotate` gives a *disjoint* annotation of the typing $\Gamma \triangleright e$, i.e. that all flow variables α_i are distinct. For a coarser analysis we could make some of the variables equal in the annotation phase, e.g. annotate all occurrences of x in $\lambda x^{\rho_1}.e^{\rho_2}$ with ρ_1 , which will decrease the number of variables and hence increase the efficiency of the analysis. This enables us to adapt our framework to finer or coarser analysis problems and to give various formal definitions of *quality of analysis*, a potential which is missing in other approaches.

Our techniques can also be used for different program analysis problems, because the type checking algorithm is essentially based on flow information that is collected through constraints. In [14] we have applied our technique to dependency analysis in the context of dead code elimination. Program slicing [38, 34], specialization [34], incrementalization [22, 23], compile-time garbage collection [18, 20, 33], and program bifurcation [30] depend on closely related notions and can thus be treated by similar techniques.

We are currently implementing our system as part of a larger project for the optimization and verification of layered communication protocols in *Nuprl* [21]. We will also consider the integration of automatic debugging and some of the program analysis problems mentioned above in this context.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *ACM Conference on Functional Programming and Computer Architecture*, pp. 31–41, 1993.
3. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *ACM SIGPLAN Conference on PLDI*, pp. 46–55, 1993.
4. R. Constable and et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

5. Patrick Cousot. Types as abstract interpretations. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 316–331, 1997.
6. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 1995. <http://www.csl.sri.com/sri-csl-fm.html>
7. D. L. Detlefs. An overview of the extended static checking system. Technical report, Digital Equipment Corporation, Systems Research Center, 1995.
8. M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. In R. M. Karp, editor, *Complexity of Computaton, SIAM-AMS Proceedings*, volume 7, pp. 27–42, 1974.
9. C. Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Computer Science Department, Rice University, 1997.
10. T. Freeman. *Refinement Types for ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1994.
11. Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.
12. D. Gries. *The Science of Programming*. Springer, 1981.
13. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 1(4):135–150, 1994.
14. O. Hafizogullari and C. Kreitz. Dead code elimination through type inference. Technical Report TR98-1673, Department of Computer Science, Cornell University, 1998.
15. S. Hayashi. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, LNCS 526, pp. 701–730. Springer, 1991.
16. N. Heintze. Set based analysis of arithmetic. Technical Report nctrl.cmu/CS-93-221, School of Computer Science, Carnegie Mellon University, 1993.
17. N. Heintze. Set-based analysis of ML programs. In *1994 ACM Conference on LISP and Functional Programming*, pp. 42–51, 1994.
18. J. Hughes. Compile-time analysis of functional programs. In D. Turner, editor, *Research Topics in Functional Programming*, chapter 5, pp. 117–153. Addison-Wesley, 1990.
19. *The Java Language Specification*.
<ftp://ftp.javasoft.com/docs/javaspec.ps.zip>
20. S.B. Jones and Metaéyer D. Le. Compile-time garbage collection by sharing analysis. In *Fourth International Conference on FPCA*, pp. 54–74, 1989.
21. C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In C. & H. Kirchner, editor, *Fifteenth International Conference on Automated Deduction*, LNAI 1421, pp. 317–332. Springer, 1998.
22. Y.A. Liu and T. Teitelbaum. Caching intermediate results for program improvement. In *ACM SIGPLAN Symposium on PEPM*, pages 190–201, 1995.
23. Y.A. Liu and T. Teitelbaum. Systematic derivation of incremental programs. *Science of Computer Programming*, 24(1):1–39, 1995.
24. D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2), 1984.
25. D.B. MacQueen. Using dependent types to express modular structure. In *13th ACM Symposium on Principles of Programming Languages*, pp. 277–286, 1986.
26. Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *ACM SIGPLAN '82 Symposium on Compiler Construction*, pp. 114–119. 1982.

27. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17):348–375, 1978.
28. J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.
29. J.C. Mitchell and R. Harper. The essence of ML. In *15th ACM Symposium on Principles of Programming Languages*, pp. 28–46, 1988.
30. T.Æ. Mogensen. Separating binding times in language specifications. In *Conference on Functional Programming Languages and Computer Architecture '89*, pp. 12–25, ACM, 1989.
31. G. C. Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages*, 1997.
32. Nuprl Project Web Page. <http://simon.cs.cornell.edu/Info/Projects/NuPr1/nuprl.html>.
33. Y.G. Park and B. Goldberg. Escape analysis on lists. In *ACM SIGPLAN Conference on PLDI*, pp. 116–127, 1992.
34. T. Reps and T. Turnidge. Program specialization via program slicing. In *Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pp. 409–429. Springer, 1996.
35. D. S. Scott. A type-theoretic alternative to CUCH,ISWIM,OWHY. *Theoretical Computer Science* 121:411-440, 1993.
36. Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.
37. Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 132–143, 1977.
38. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.
39. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on PLDI*, 1998.