Automating Relatively Complete Verification of Higher-Order Functional Programs

Hiroshi Unno
University of Tsukuba
uhiro@cs.tsukuba.ac.jp

Tachio Terauchi
Nagoya University
terauchi@is.nagoya-u.ac.jp

Naoki Kobayashi
University of Tokyo
koba@is.s.u-tokyo.ac.jp

Abstract

We present an automated approach to relatively completely verifying safety (i.e., reachability) property of higher-order functional programs. Our contribution is two-fold. First, we extend the refinement type system framework employed in the recent work on (incomplete) automated higher-order verification by drawing on the classical work on relatively complete "Hoare logic like" program logic for higher-order procedural languages. Then, by adopting the recently proposed techniques for solving constraints over quantified first-order logic formulas, we develop an automated type inference method for the type system, thereby realizing an automated relatively complete verification of higher-order programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Relative Completeness, Higher-Order Programs, Software Model Checking, Type Inference

1. Introduction

Recently, motivated by the success of *software model checkers* [12] for the automated verification of first-order programs, researchers have proposed "model checkers" for the automated verification of higher-order programs [13, 18, 23, 26, 27]. Interestingly, they have all been formulated as a form of refinement type inference.[1]

The refinement type systems employed in the automated higher-order program verification have some important differences from the ones from the original, non-automated approaches like DML [30], making them more amenable to automation (such as the lack of implicit Π types). But, they follow the non-automated ones in that the types embed first-order logic (FOL) formulas, called *refinement predicates*, over program values (as in dependent types) that are used to express and enforce detailed properties of the program.

For example, consider the OCaml program shown in Figure 1. Here, `*` denotes a non-deterministic choice. Given an integer argument i to `main`, the program recursively calls `app` non-deterministically to apply the closure `check` i to $i + j$ where j is the number of times the then branch is taken in `app`. The program is safe in that `assert false` is unreachable for any argument i. The program is difficult to verify via first-order program verification methods because of the higher-order recursive function `app`.

```
let rec app x f = if * then app (x+1) f
                  else f x in
let check x y = if x <= y then ()
                else assert false in
let main i = app i (check i)
```

Figure 1. A simple higher-order program.

Recent advances in higher-order program verification have enabled automated verification of such programs via refinement type inference. For the program above, the following refinement types may be automatically inferred to verify its safety.[2]

$$\begin{aligned} \texttt{app} &: x{:}\texttt{int} \to f{:}(\{u{:}\texttt{int} \mid u \geq x\} \to \texttt{unit}) \to \texttt{unit} \\ \texttt{check} &: x{:}\texttt{int} \to y{:}\{u{:}\texttt{int} \mid u \geq x\} \to \texttt{unit} \\ \texttt{main} &: i{:}\texttt{int} \to \texttt{unit} \end{aligned}$$

The type of `app` expresses the fact that the function takes an integer argument x, and a function-type argument f which takes an integer at least as large as x. The type of `check` says that it takes integers x and y such that $x \leq y$. And, the type of `main` says that it takes any integer argument.

The refinement type systems underlying the verifiers are sound, that is, they only type safe programs. However, they are incomplete in that there are safe programs that they cannot type. Indeed, the only known positive result is for the class of the finite domain base-type data programs, which can be verified completely by a refinement type system augmented with intersection types [16, 17]. The situation is in stark contrast to that of automated first-order program verification [12] where the underlying program logic, such as the Hoare logic, is relatively complete.

For example, none of the refinement type systems proposed for automated higher-order program verification can type and verify the program shown in Figure 2, even though the program is only a small modification of the one from Figure 1: it simply switches the order of `app`'s arguments. Nor, are they able to type and verify the program shown in Figure 3, which uses the function `succ` to successively build closures to pass integers larger than `i` to `check i`. The program is untypable even if we were allowed to change the order of the function arguments.

We note that this is an incompleteness at the level of the program logic (i.e., refinement type system) and not the verification algorithm (i.e., type inference algorithm). That is, there exist no types within the refinement type system that can type the program,

[1] In fact, the equivalence of model checking and refinement type inference has been shown for the finite domain base-type data case [16, 17].

[2] The type syntax is borrowed from Augustsson [1].

```
let rec app f x = if * then app f (x+1)
                  else f x in
let check x y = if x <= y then ()
                else assert false in
let main i = app (check i) i
```

Figure 2. A variant of Figure 1.

```
let rec app3 f g = if * then app3 (succ f) g
                   else g f in
let app x f = f x in
let succ f x = f (x + 1) in
let check x y = if x <= y then ()
                else assert false in
let main i = app3 (check i) (app i)
```

Figure 3. Another variant of Figure 1.

let alone inferable ones. While higher-order program verification is in general undecidable, like the first-order program verification is, it would be desirable to have a relatively complete reasoning framework that would serve as the basis of verification algorithms, as the Hoare logic does for first-order programs.

To this end, this paper presents an automated approach to a relatively complete verification of higher-order programs. First, we present an extension to the refinement type system such that the resulting refinement type system is relatively complete. The extension is inspired by the classical work on relatively complete program logic for higher-order procedural languages by German et al. [6, 7] (also, [8]) who showed that relative completeness is achievable while avoiding the explicit use of functions as data so as to maintain a "Hoare logic like" control-data separation.

However, their proof of relative completeness does use encoding of functions as base-type data, and we rely on the same technique for our relative completeness proof. Moreover, to hide the use of functions as data from the program logic, they introduce quantifiers that the client verifier must appropriately instantiate. (To the best of our knowledge, no actual verifier was built based on their program logic.) Therefore, as our second contribution, we show a type inference method that extends the first-order logic constraint solving of the previous automated refinement type inference systems to quantified reasoning by leveraging recent advances on a related problem [4, 9, 24].

In summary, the paper's contributions are as follows.

- A refinement type system that is relatively complete for safety verification of higher-order functional programs.
- A type inference algorithm for the refinement type system.

In the next section, we give an informal overview of the main ideas.

On incompleteness of the inference: The type inference algorithm that we propose is, of course, incomplete in that it is not able to decide the typability of all programs. This is expected because safety verification is undecidable in general. Instead, our contribution is a type system that is complete relative to a hypothetical theorem prover complete for first-order arithmetic, and a novel inference algorithm that is able to automatically verify a non-trivial subset of the programs that were not possible to verify with the previous automated approaches.

2. Informal Overview

We informally describe the incompleteness issue by showing how the existing approaches [13, 18, 23, 26, 27] fail to type the programs shown in Figures 2 and 3.

First, let us try to type Figure 2. Here, the goal is to show that `main` can be given the type $i{:}\texttt{int} \to \texttt{unit}$, that is, `main` is safe to be called with any integer i. (For a base type B, we often abbreviate the refinement type $\{u{:}B \mid \top\}$ as B. E.g., $\texttt{int} = \{u{:}\texttt{int} \mid \top\}$.) Therefore, we try to type the body of `main` under the assumption that the type of `i` is $\{u{:}\texttt{int} \mid \top\}$. The most precise type for the partial application `check i` is $y{:}\{u{:}\texttt{int} \mid u \geq \texttt{i}\} \to \texttt{unit}$ (i.e., functions that can take any integer at least as large as `i`), and the most precise type for `i` is $\{u{:}\texttt{int} \mid u = \texttt{i}\}$ (i.e., integers equal to `i`). (Intuitively, $\{u{:}B \mid \theta\}$ expresses values of the base type B satisfying the refinement predicate θ, and $x : \tau \to \sigma$ expresses functions that return a value of the type $\sigma[e/x]$ when given the argument e of the type τ.)

We show that the type systems fail to give the higher-order function `app` a sufficiently precise type to verify the program's safety. The type of `app` must be of the form

$$f{:}(\{u{:}\texttt{int} \mid \theta\} \to \texttt{unit}) \to x{:}\{u{:}\texttt{int} \mid \phi\} \to \texttt{unit}$$

where the refinement predicates θ and ϕ are FOL formulas in the theory of base-type data.[3] In addition, as discussed below, to prevent degenerate types, the refinement type systems enforce an important *well-formedness* condition that restricts the variables that can appear free in a refinement predicate. (Here, θ and ϕ are restricted so that $\mathit{fv}(\theta) \subseteq \{u\}$ and $\mathit{fv}(\phi) \subseteq \{f, u\}$, where $\mathit{fv}(\theta)$ denotes the free variables of θ.)

From the application `f x` in the body of `app`, the type system asserts that $\phi \Rightarrow \theta$, that is, `f` must be safe to be given the argument `x`. In the body of `main`, `app` is applied to `check i` and `i`, and the type system asserts that the type of `check i` is a subtype of $\{u{:}\texttt{int} \mid \theta\} \to \texttt{unit}$ and the type of `i` is a subtype of $\{u{:}\texttt{int} \mid \phi\}$. This leads to the constraints $u = \texttt{i} \Rightarrow \phi$ and $\theta \Rightarrow u \geq \texttt{i}$, accurately expressing the fact that `i` is passed as the second argument to `app` and that the first argument to `app` (i.e., `check i`) expects a value at least as large as `i`.

A solution to the set of constraints is $\theta \equiv u \geq \texttt{i}$ and $\phi \equiv u = \texttt{i}$, but this is degenerate because `main`'s variable `i` would appear in `app`'s type. (Recall that θ, ϕ are refinement predicates of `app`'s type.) Intuitively, it means that the function `app` could "see" values from a specific context of its use (i.e., `main`). Indeed, as remarked above, the refinement type systems disallow such degenerate types via the well-formedness condition, and only allow θ, ϕ such that $\mathit{fv}(\theta) \subseteq \{u\}$ and $\mathit{fv}(\phi) \subseteq \{f, u\}$. That is, the refinement predicates are allowed to only mention the variables in their respective scopes. Consequently, the program is (conservatively) rejected as untypable. It is worth noting that the program of Figure 1 does not have this issue as `app`'s arguments are conveniently ordered so that `app`'s function-type argument could depend on the base-type argument to allow the following sufficiently precise type:

$$x{:}\texttt{int} \to f{:}(\{u{:}\texttt{int} \mid u \geq x\} \to \texttt{unit}) \to \texttt{unit}$$

Unfortunately, as exemplified by Figure 3, the incompleteness issue is not just a matter of choosing the right order of function arguments. Here, the program is untypable even if we were allowed to change the order of the arguments, and the untypability comes from not being able to give a precise enough type to `app3`. Ideally, we would like to express via `app3`'s type the fact that `g` is safe when applied to `f` (and also when applied to $\texttt{succ}^j\ \texttt{f}$ for any $j > 0$). But, this requires a type that is parametric in `f` and `g`'s behavior, and that cannot be given because the only parameters of `app3` are `f` and `g`, and they are both function-type arguments. Note that refinement predicates of the form "`g` is safe to be called with `f`" are prohibited as they are required to be FOL formulas over the base-type data.

[3] We could also give non-$\top$ refinement predicates to `unit`, but it does not affect the example.

2.1 Our Approach

Our approach to solving the incompleteness issue is inspired by the research on relatively complete program logic for higher-order procedural languages [6–8]. The main idea is to add extra *dummy* base-type parameters that are *instantiated* appropriately so that the refinement type of a higher-order function can depend on the parameters.

For example, for the program of Figure 2, we add an extra base-type parameter `a` to `app` and obtain the following program.

```
let rec app a f x = if * then app[a] f (x+1)
                    else f x in
let check x y = if x <= y then ()
                else assert false in
let main i = app[i] (check i) i
```

Here, for clarity, the parameter passings for the extra parameters are written as quantifier instantiations, but they may be understood as ordinary function applications (e.g., `app[a]` is `app a`). With the addition of the extra parameter, it becomes possible for the existing refinement type systems designed for automated verification to type and verify the program. For example, the following types are sufficient for typing the program.

$$\begin{aligned}\texttt{app} &: a{:}\texttt{int} \to f{:}(\tau \to \texttt{unit}) \to x{:}\tau \to \texttt{unit}\\ \texttt{check} &: x{:}\texttt{int} \to y{:}\{u{:}\texttt{int} \mid u \geq x\} \to \texttt{unit}\\ \texttt{main} &: i{:}\texttt{int} \to \texttt{unit}\end{aligned}$$

where $\tau = \{u{:}\texttt{int} \mid u \geq a\}$. Note that the extra parameter a is used to parametrize f's behavior in the type of `app`.

There is a simple rule to adding extra parameters that can be shown to be sufficient for relative completeness: *add one just before each function-type argument*. Following the rule, for example, the program of Figure 3 is translated as follows.

```
let rec app3 a f b g =
              if * then (app3[a] (succ[a] f))[a] g
              else g[a] f in
let succ b f x = f (x + 1) in
let check x y = if x <= y then ()
                else assert false in
let app x a f = f x in
let main i = (app3[i] (check i))[i] (app i)
```

And, it can be shown that the resulting program is typable, for example, by the following types.

$$\begin{aligned}\texttt{app3} &: a{:}\texttt{int} \to f{:}\tau_1 \to b{:}\texttt{int} \to g{:}\tau_2 \to \texttt{unit}\\ \texttt{app} &: \\ & \quad x{:}\texttt{int} \to a{:}\texttt{int} \to f{:}(\{u{:}\texttt{int} \mid u \geq x\} \to \texttt{unit}) \to \texttt{unit}\\ \texttt{succ} &: b{:}\texttt{int} \to f{:}(\sigma \to \texttt{unit}) \to x{:}\sigma \to \texttt{unit}\\ \texttt{check} &: x{:}\texttt{int} \to y{:}\{u{:}\texttt{int} \mid u \geq x\} \to \texttt{unit}\\ \texttt{main} &: i{:}\texttt{int} \to \texttt{unit}\end{aligned}$$

where $\sigma = \{u{:}\texttt{int} \mid u \geq b\}$, $\tau_1 = \{u{:}\texttt{int} \mid u \geq a\} \to \texttt{unit}$, and $\tau_2 = c{:}\texttt{int} \to \tau_1 \to \texttt{unit}$.

However, the rule does not answer how the extra parameters should be instantiated. (In the above, we seem to have magically conjured the appropriate instantiations `i` and `a`!) Indeed, an analogous rule was first discovered by German, Clarke, and Halpern [6, 7] in their work on a relatively complete program logic for higher-order procedural languages, and their proof of relative completeness relies on the fact that, with an expressive theory of the base-type data domain (such as Peano arithmetic), one can encode each function closure as a base-type data expression so that the extra parameters can be instantiated by the base-type data representation of the corresponding function-type argument. However, explicitly instantiating the extra parameters by such encoded expressions and forcing the client verifier to reason about them is impractical.

$$\begin{array}{rcl} d & ::= & d \cup \{F\ \overrightarrow{x}\ \texttt{=}\ e\} \mid \emptyset \\ e & ::= & x \mid F \mid c \mid \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2 \mid e\ x \\ & \mid & \texttt{if} * \texttt{then}\ e_1\ \texttt{else}\ e_2 \end{array}$$

Figure 4. The syntax of the simple functional language.

Therefore, they proposed to leave the instantiations unspecified in the program logic so that the task of finding sufficient instantiations is left to (the FOL theorem prover of) the client verifier.

Following the idea, for example, the program of Figure 2 is translated as follows by introducing fresh variables `v1` and `v2`.

```
let rec app a f x = if * then app[v1] f (x+1)
                    else f x in
let check x y = if x <= y then ()
                else assert false in
let main i = app[v2] (check i) i
```

Then, the refinement type inference problem is reduced to the problem of finding appropriate instantiations for `v1` and `v2` along with the types for the program. For this, we extend the FOL constraint solving used in the previous work on refinement type inference with quantified reasoning over *template* expressions [4, 9, 24]. Concretely, in Section 4, we show how the counterexample-guided refinement type inference [18, 26] can be extended in this way to infer appropriate instantiations as well as types.

2.2 Paper Organization

The rest of the paper is organized as follows. Section 3 presents the refinement type system along with the target functional language, and proves its relative completeness under the extra parameter addition strategy. Section 4 presents the type inference algorithm. Section 5 presents the preliminary experience with the type inference algorithm, implemented as an extension to the higher-order software model checker (i.e., refinement type inference system) MoCHi [18]. Section 6 discusses related work, and Section 7 concludes. The extended report [28] contains the omitted materials.

3. Language and Type System

We formalize the target programming language. Figure 4 shows the syntax. The language is essentially the simply-typed lambda calculus with recursion and primitives for integer arithmetic operations. For simplicity the only base-type data in this language is integers, but other base types and their operations can be encoded in the standard way: for example, $\mathsf{true} = 1$ and $\mathsf{false} = 0$.

A *program*, d, is a finite set of function definitions, $F\ \overrightarrow{x}\ \texttt{=}\ e$, defining a function named F with the formal parameters $\overrightarrow{x}$ and the body e. The notation $\overrightarrow{a}$ denotes a possibly empty sequence. The functions are mutually recursive in that the body of a function may refer to other functions, including itself. Each function is closed except for the free function names (i.e., functions are lambda lifted [15]). We also assume that d contains a function named `main` that only takes base-type (i.e., integer) arguments.

Expressions, e, comprise non-deterministic branches, let expressions, constants c, function names F, variables x, and (constant or user-defined function) applications $e\ x$. Constants include integer constants such as $-1, 0, 1, 2$, and integer operations such as $+$ and $\leq$ (recall that we model booleans via integers). We assume that there are unary constant operators named `assert` and `assume`.

We restrict expressions to be in continuation passing style (CPS) so that they are non-returning, except when they occur let-bound (i.e., occurs as e_1 in $\texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2$). By contrast, let-bound expressions are restricted to be non-CPS (i.e., value returning) expressions, which are total applications of constant operators including the assume and assertion expressions $\texttt{assume}\ x$ and $\texttt{assert}\ x$, and

$$
\begin{array}{l}
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to_d e_2[e_3/x] \quad \text{where } e_1 \Downarrow e_3 \\
\texttt{let}\ x = \texttt{assert}\ 1\ \texttt{in}\ e \to_d e[1/x] \\
\texttt{let}\ x = \texttt{assert}\ i\ \texttt{in}\ e \to_d \mathsf{fail} \quad \text{where } i \neq 1 \\
\texttt{let}\ x = \texttt{assume}\ 1\ \texttt{in}\ e \to_d e[1/x] \\
\texttt{let}\ x = \texttt{assume}\ i\ \texttt{in}\ e \to_d \mathsf{safe} \quad \text{where } i \neq 1 \\
\texttt{if} * \texttt{then}\ e_1\ \texttt{else}\ e_2 \to_d e_1 \\
\texttt{if} * \texttt{then}\ e_1\ \texttt{else}\ e_2 \to_d e_2 \\
F\ \overrightarrow{e} \to_d e'[\overrightarrow{e}/\overrightarrow{x}] \quad \text{where } F\ \overrightarrow{x} = e' \in d
\end{array}
$$

Figure 5. The reduction rules.

partial applications (of user-defined functions). We restrict constant operator applications to be total. CPS/A-normal-form-style is used for simplicity. Direct-style syntax can be supported by CPS conversion.

The rest of the syntax is straightforward. As usual, applications associate to the left so that $e_0\ e_1\ e_2 = (e_0\ e_1)\ e_2$. We write $e_0\ \overrightarrow{e}$ for the series of applications $e_0\ e_1\ e_2\ \ldots\ e_n$ where $\overrightarrow{e} = e_1, e_2, \ldots, e_n$. We write $e_1; e_2$ for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ such that $x \notin \mathit{fv}(e_2)$, where $\mathit{fv}(e)$ denotes the free variables of e. Without loss of generality, we assume that bound variables are distinct. Note that, while the language only has non-deterministic branches, a conditional branch $\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ can be encoded as

$$\texttt{if} * \texttt{then}\ \texttt{assume}\ x; e_1\ \texttt{else}\ \texttt{let}\ y = \neg x\ \texttt{in}\ \texttt{assume}\ y; e_2$$

which is equivalent for assertion safety.

We define the operational semantics of the language as a small-step reduction relation from states to states. A *state* is a run-time expression e that extends the source expressions with non-variable arguments at function applications, a special failure state fail, and a special safe state safe. (We overload the symbol e to range over run-time expressions when it is clear from the context.)

The reduction relation $\to_d$ is defined by the rules shown in Figure 5. Here, $e \Downarrow e'$ denotes the evaluation of the value returning expression e to the *value* e', and is defined as $F\ \overrightarrow{e} \Downarrow F\ \overrightarrow{e}$ and $c\ \overrightarrow{e} \Downarrow [\![c]\!](\overrightarrow{e})$ where $[\![c]\!]$ is the relation denoting the semantics of a non-assume/assert constant c, so that, for example $[\![+]\!](i, j) = i + j$ for integers i and j. (An integer constant is represented by a 0-ary constant.) We let partial applications (i.e., $F\ \overrightarrow{e}$) represent function closures. Formally, a value is either an integer constant or a function closure.

The semantics of a program d is defined as a series of reductions starting from an initial state $e_{\texttt{main}}[\overrightarrow{i}/\overrightarrow{x}]$ where $\texttt{main}\ \overrightarrow{x} = e_{\texttt{main}} \in d$ and $\overrightarrow{i}$ are integer arguments for main (i.e., $|\overrightarrow{i}| = |\overrightarrow{x}|$). Note that, because of CPS, reductions only occur at the top level. We write $e \to_d^* e'$ for zero or more reductions from e to e'. We say that a program is *(assertion) safe* if its evaluation does not cause an assertion failure, that is, if $e_{\texttt{main}}[\overrightarrow{i}/\overrightarrow{x}] \not\to_d^* \mathsf{fail}$ for any arguments $\overrightarrow{i}$ of main.

We assume that a program is typed under the standard simple type system whose type grammar is shown below.

$$s ::= \star \mid \texttt{int} \mid s \to s'$$

Here, the type $\star$ represents the type of a CPS expression. For each expression e in the program, we write $\mathit{sty}(e)$ to denote its simple type.

The typability in the simple type system assures that the program does not "get stuck", for example, by trying to use an integer as a function, but it does not guarantee its safety. Therefore, a program either runs forever safely (due to CPS, a program cannot return), stops safely in the state safe by reaching a false assume, or aborts with an assertion failure. The typing rules for the simple type system are standard and are deferred to the extended report [28].

$$\tau, \sigma \quad ::= \quad \star \mid \{u \mid \theta\} \mid x{:}\sigma \to \tau$$

Figure 6. The syntax of refinement types.

$$\frac{\mathit{sty}(x) = \texttt{int}}{\Gamma \vdash x : \{u \mid u = x\}}\ \mathbf{Vb} \qquad \frac{\mathit{sty}(\kappa) \in \to}{\Gamma \vdash \kappa : \Gamma(\kappa)}\ \mathbf{Vf}$$

$$\frac{}{\Gamma \vdash c : \mathit{ty}(c)}\ \mathbf{Cst} \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x{:}\sigma \vdash e_2 : \star}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \star}\ \mathbf{Let}$$

$$\frac{\Gamma \vdash e : y{:}\sigma \to \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash e\ x : \tau[x/y]}\ \mathbf{App}$$

$$\frac{\Gamma \vdash e_1 : \star \quad \Gamma \vdash e_2 : \star}{\Gamma \vdash \texttt{if} * \texttt{then}\ e_1\ \texttt{else}\ e_2 : \star}\ \mathbf{If}$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \leq \tau \quad \mathit{fv}(\tau) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash e : \tau}\ \mathbf{Sub}$$

Figure 7. The typing rules.

3.1 Refinement Type System

We present a refinement type system for the language. First, we present a sound but incomplete refinement type system without the extra parameter addition, and then introduce the extra parameter addition as an extension in Section 3.2. The (incomplete) refinement type system is not significantly different from the ones proposed previously for automated verification [13, 18, 23, 26, 27] (e.g., it can be obtained by removing intersection types from [18, 26][4]).

Figure 6 shows the syntax of refinement types. Here, $\{u \mid \theta\}$ is a *refinement base (i.e., integer) type* that refines an integer by the refinement predicate θ which is a FOL arithmetic formula over the base (i.e., integer) type variables.[5] We sometimes abbreviate $\{u \mid \theta\}$ as int when θ is a tautology (e.g., $\{u \mid \top\} = \texttt{int}$). Intuitively, $\{u \mid \theta\}$ denotes the type of integers u satisfying the formula θ.

The type $x{:}\sigma \to \tau$ is a *dependent function type*, consisting of the argument type σ and the return type τ. Intuitively, $x{:}\sigma \to \tau$ denotes the type of a function (or a constant operation) that returns a value of the type $\tau[y/x]$ when applied to an argument y of the type σ. As usual, $\to$ associates to the right.

The type $\{x \mid \theta\}$ binds x in θ. Likewise, $x{:}\sigma \to \tau$ binds x in τ (but not in σ). That is,

$$\begin{array}{l}
\mathit{fv}(\{x \mid \theta\}) = \mathit{fv}(\theta) \setminus \{x\} \\
\mathit{fv}(x{:}\sigma \to \tau) = \mathit{fv}(\sigma) \cup (\mathit{fv}(\tau) \setminus \{x\})
\end{array}$$

We sometimes abbreviate $x{:}\sigma \to \tau$ as $\sigma \to \tau$ when x does not occur free in τ. (It actually suffices to limit x to occur free in τ only if x is a base type argument because refinement predicates are restricted to be over the base-type variables.) Types are equivalent up to renaming of bound variables.

The *simple-type shape* of σ, $\mathit{tshape}(\sigma)$, is defined as follows:

$$\begin{array}{l}
\mathit{tshape}(\{u \mid \theta\}) = \texttt{int} \qquad \mathit{tshape}(\star) = \star \\
\mathit{tshape}(x{:}\sigma \to \tau) = \mathit{tshape}(\sigma) \to \mathit{tshape}(\tau)
\end{array}$$

[4] While intersection types are not needed for relative completeness, they compromise neither soundness nor completeness, and sometimes aid the verification in practice. The implementation shown in Section 5 supports intersection types.

[5] The full theory of Peano arithmetic is needed for relative completeness, but any subset (e.g., the quantifier-free theory of linear arithmetic) is sufficient for soundness.

$$\frac{\Gamma \vdash \sigma_2 \leq \sigma_1 \quad \Gamma, x:\sigma_2 \vdash \tau_1 \leq \tau_2}{\Gamma \vdash x{:}\sigma_1 \to \tau_1 \leq x{:}\sigma_2 \to \tau_2}\ \mathbf{Sf}$$

$$\frac{}{\Gamma \vdash \star \leq \star}\ \mathbf{Sc} \qquad \frac{u \notin fv([\![\Gamma]\!]) \quad ([\![\Gamma]\!] \wedge \theta_1) \Rightarrow \theta_2}{\Gamma \vdash \{u \mid \theta_1\} \leq \{u \mid \theta_2\}}\ \mathbf{Sb}$$

Figure 8. The subtyping rules.

Figure 7 shows the typing rules. The judgements for the expressions are of the form $\Gamma \vdash e : \tau$ where Γ is a *type environment* mapping variables and function names to types.

We discuss each typing rule. **Vb** types base-type variables. Note that the rule ignores the environment. Expressibility is not reduced, however, because the assumption about x in the environment gets discharged at subtyping. **Vf** types function-type variables and function names by looking up the environment. Here, $\to$ denotes the set of simple function types. (We use the meta variable κ to range over both variables and function names.) **Cst** types constants. Here, $ty(c)$ denotes the pre-assigned type of the constant c such that

$$ty(\texttt{assume}) = x{:}\texttt{int} \to \{u \mid x = 1 \wedge u = 1\}$$
$$ty(\texttt{assert}) = x{:}\{u \mid u = 1\} \to \{u \mid u = 1\}$$

and $ty(c)$ is the *precise* type for a non-assume/assert constant c (e.g., $ty(+) = x:\texttt{int} \to y{:}\texttt{int} \to \{u \mid u = x + y\}$).[6] **Let** is self-explanatory. **App** types applications. Here, $\tau[x/y]$ is the usual capture-avoiding substitution. **If** types branch expressions.

Sub is the subsumption rule. The subtyping relation is defined in Figure 8. In **Sb**, $[\![\Gamma]\!]$ is the FOL formula denoting the assumptions about the base-type variables in Γ, and is defined as follows.

$$[\![\Gamma]\!] = \bigwedge\nolimits_{x:\{u|\theta\} \in \Gamma} \theta[x/u]$$

Like the *rule of consequence* of the Hoare logic [10], **Sb** asks the client verifier (i.e., the type inference system) to decide the validity of arbitrary FOL arithmetic formulas. As expected, our completeness result is *relative* to the hypothetical completeness of deciding this. In verification practice, one may settle for incomplete theorem proving or a decidable theory subset to make type checking decidable.[7]

Sub enforces *well-formedness* of the introduced type by asserting that its free variables appear bound in the environment. As shown below, the top-level types (i.e., the types of the recursive functions) are required not to contain free variables.

DEFINITION 3.1 (Well-formed type). *We say that a type is* closed *if it has no free variables. Let Δ be a top-level type environment mapping function names to types. We say that σ is a* well-formed type *for F if σ is closed and $tshape(\sigma) = sty(F)$. We say that Δ is a* well-formed top-level type environment *if $\Delta(F)$ is well-formed for each F.*

Let us write $\overrightarrow{x{:}\sigma} \to \tau$ to abbreviate the function type $x_1{:}\sigma_1 \to \cdots \to x_n{:}\sigma_n \to \tau$ where $\overrightarrow{x{:}\sigma} = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$. We define the notion of a well-typed program.

DEFINITION 3.2 (Well-typed program). *We write $\Delta \vdash d$ if the following conditions hold.*

(1) Δ is a well-formed top-level type environment.

(2) For each function $F\ \overrightarrow{x} = e \in d$, we have $\Delta, \overrightarrow{x{:}\sigma} \vdash e : \star$ where $\Delta(F) = \overrightarrow{x{:}\sigma} \to \star$.

(3) $\Delta(\texttt{main})$ is of the form $\overrightarrow{x{:}\texttt{int}} \to \star$. (I.e., the refinement predicates for the arguments of `main` *are all $\top$.)*

A program d is said to be well-typed (equivalently, typable) if there exists Δ such that $\Delta \vdash d$.

The condition (2) says that Δ contains fixed-point types for the recursive functions comprising d, and (3) says that `main` is safe to be called with any arguments (recall that `main` is a function over integer arguments).

The type system is sound in that it ensures that a well-typed program does not cause an assertion failure.

THEOREM 3.3 (Soundness). *If $\Delta \vdash d$ then d is safe.*

The theorem follows from the soundness of the refinement type system extended with extra parameter additions (Theorem 3.6).

EXAMPLE 3.4. Let the program d_1 consist of the following functions.

```
app x f = if * then app (x + 1) f else f x
check x y = if * then assume (x ≤ y); check x y
            else assume ¬(x ≤ y); assert 0; check x y
main i = app i (check i)
```

The program is the one from Figure 1 translated into the target language. (We elide A-normalization for readability.) Let Δ be the following type environment.

$$\Delta(\texttt{app}) = x{:}\texttt{int} \to f{:}(\{u \mid u \geq x\} \to \star) \to \star$$
$$\Delta(\texttt{check}) = x{:}\texttt{int} \to y{:}\{u \mid u \geq x\} \to \star$$
$$\Delta(\texttt{main}) = i{:}\texttt{int} \to \star$$

It is routine to check that $\Delta \vdash d_1$. Therefore, d_1 is typable and is safe.

Next, consider d_2 shown below.

```
app f x = if * then app f (x + 1) else f x
check x y = if * then assume (x ≤ y); check x y
            else assume ¬(x ≤ y); assert 0; check x y
main i = app (check i) i
```

The program is a translation of the program from Figure 2. The program is safe but untypable (as remarked in Section 2, also under the previous refinement type systems [13, 18, 23, 26, 27]). Note that we cannot simply assign `app` the type $\Delta(\texttt{app})$ from the above but with the order of x and f reversed so that

$$f{:}(\{u \mid u \geq x\} \to \star) \to x{:}\texttt{int} \to \star$$

The type is not closed and therefore is not a well-formed top-level type (cf. Definition 3.1). The well-formedness condition forces the refinement predicates to only refer to the values passed earlier. Well-formedness is not a superficial restriction: In the presence of higher-order functions and function closures (i.e., partial applications), we cannot generally determine "up front" in the program logic what will be passed later to a closure.

The issue is apparent in the following program d_3, which is a translation of Figure 3.

```
app3 f g = if * then app3 (succ f) g else g f
app x f = f x
succ f x = f (x + 1)
check x y = if * then assume (x ≤ y); check x y
            else assume ¬(x ≤ y); assert 0; check x y
main i = app3 (check i) (app i)
```

As remarked in Section 2, the program is safe but untypable, even if we were allowed to change the order of the function arguments. Here, it is not possible to determine what will be passed to the

[6] See the extended report [28] for the definition of a precise constant type. Also, non-precise but *sound* constant types, that over-approximate the actual semantics, are sufficient for soundness.

[7] Also, type checking being decidable is different from type inference (i.e., typability) being decidable. The previous research has proposed various incomplete inference approaches [13, 18, 23, 26, 27].

$$\frac{\Gamma \vdash_\forall e : \forall x{:}\sigma.\tau \quad \Gamma \vdash_\forall o : \sigma \quad o \in \mathit{pureExps}}{\Gamma \vdash_\forall e : \tau[o/x]}\ \mathbf{Inst}$$

$$\frac{\Gamma \vdash_\forall \sigma_2 \leq \sigma_1 \quad \Gamma, x{:}\sigma_2 \vdash_\forall \tau_1 \leq \tau_2}{\Gamma \vdash_\forall \forall x{:}\sigma_1.\tau_1 \leq \forall x{:}\sigma_2.\tau_2}\ \mathbf{S}\forall$$

Figure 9. Additional typing rules.

closures $\texttt{succ}^j\ f$ without some non-trivial program reasoning (i.e., i captured in g added the number of times `app3`'s then branch is taken before the else branch is reached). In effect, the idea of the extra parameter addition for relative completeness is to delegate such tasks to the client verifier.

3.2 Extra Parameters for Relative Completeness

As shown in Example 3.4 (also Section 2), the refinement type system is incomplete by itself. While a complete checking of safety is clearly undecidable as the language allows arbitrary integer operations, we would like to make the refinement type system be complete relative to an oracle that could decide the FOL implications discharged at the subtyping rule **Sb**.

Rather than formulating the extra parameter addition as a program translation as done in Section 2, here, we present it as a type system extension in the form of universally quantified types. This is expositionally convenient, because extra parameters can be represented as universally bound variables so that they can be easily distinguished from ordinary parameters.

We extend the grammar of refinement types with universally quantified types as follows.

$$\tau, \sigma ::= \ldots \mid \forall x{:}\sigma.\tau$$

In $\forall x{:}\sigma.\tau$, x is a binding occurrence and may occur free in τ (i.e., $\mathit{fv}(\forall x{:}\sigma.\tau) = \mathit{fv}(\sigma) \cup (\mathit{fv}(\tau) \setminus \{x\})$). Intuitively, it expresses the function type $x{:}\sigma \to \tau$ such that x is an extra parameter. We define $\mathit{tshape}(\forall x{:}\sigma.\tau) = \mathit{tshape}(\tau)$.

We extend the type system with the quantifier instantiation and the subtyping rules shown in Figure 9. Here, $\mathit{pureExps}$ is the set of side-effect-free integer-type expressions defined by the grammar

$$o ::= x \mid c\ \overrightarrow{o}$$

where x is an integer-type variable and $c\ \overrightarrow{o}$ is a total application of a non-assume/assert constant.[8] Therefore, **Inst** allows instantiation with any in-scope side-effect-free integer-type expression having the requested type. **S**$\forall$ is analogous to the subtyping rule for function types (i.e., **Sf**). To distinguish, we write $\vdash_\forall$ for the type judgements of the extended type system. All the rules from Figure 7 and Figure 8 are assumed to be included in $\vdash_\forall$ (with $\vdash$ replaced by $\vdash_\forall$).

The well-formedness definition is also retained from the pre-extension (cf. Definition 3.1). Therefore, for example, both

$$x{:}\texttt{int} \to (\{u \mid u = 0\} \to \star) \to \star$$
$$x{:}\texttt{int} \to \forall y{:}\texttt{int}.(\{u \mid u = y\} \to \star) \to \star$$

are well-formed refinement types for F such that $\mathit{sty}(F) = \texttt{int} \to (\texttt{int} \to \star) \to \star$.

Let γ be a disjoint union of variable bindings $x{:}\sigma$ or *universal quantifier bindings* $\forall x{:}\sigma$. The type abbreviation $\overrightarrow{\gamma} \to \tau$ is defined as follows.

$$\overrightarrow{\gamma} \to \tau = \begin{cases} \tau & \text{if } \overrightarrow{\gamma} = \varepsilon \\ \forall x{:}\sigma.\tau_1 & \text{if } \overrightarrow{\gamma} = \forall x{:}\sigma, \overrightarrow{\gamma_1} \text{ and } \overrightarrow{\gamma_1} \to \tau = \tau_1 \\ x{:}\sigma \to \tau_1 & \text{if } \overrightarrow{\gamma} = x{:}\sigma, \overrightarrow{\gamma_1} \text{ and } \overrightarrow{\gamma_1} \to \tau = \tau_1 \end{cases}$$

[8] We extend the typing rules to (non-A-normal-form) side-effect-free integer-type expressions in the obvious way. See the extended report [28].

(Note that this subsumes the $\overrightarrow{x{:}\sigma} \to \tau$ abbreviation introduced earlier.) Let $\langle\overrightarrow{\gamma}\rangle$ be the sequence $\overrightarrow{\gamma}$ with each universal quantifier binding $\forall x : \sigma$ replaced with the variable binding $x : \sigma$. That is, $\langle\varepsilon\rangle = \varepsilon$, and $\langle x{:}\sigma, \overrightarrow{\gamma}\rangle = \langle \forall x{:}\sigma, \overrightarrow{\gamma}\rangle = x{:}\sigma, \langle\overrightarrow{\gamma}\rangle$.

We extend the notion of well-typed program to accommodate universal quantifier bindings.

DEFINITION 3.5 (Well-typed program – Extended). *We write $\Delta \vdash_\forall d$ if the following conditions hold.*

(1) Δ is a well-formed top-level type environment.
(2) For each function $F\ \overrightarrow{x}\ \texttt{=}\ e \in d$, we have $\Delta, \langle\overrightarrow{\gamma}\rangle \vdash_\forall e : \star$ where $\Delta(F) = \overrightarrow{\gamma} \to \star$.
(3) $\Delta(\texttt{main})$ is of the form $\overrightarrow{x{:}\texttt{int}} \to \star$.

A program d is said to be well-typed if $\exists\Delta.\Delta \vdash_\forall d$.

Note that the only differences from Definition 3.2 are $\overrightarrow{\gamma} \to \star$ and $\langle\overrightarrow{\gamma}\rangle$ in (2) which allow universal quantifiers in the type of the functions. Also, note that the well-formedness condition now allows the refinement predicates to refer to the universally quantified variables in their scope.

We state the soundness of the extended refinement type system.

THEOREM 3.6 (Soundness – Extended). *If $\Delta \vdash_\forall d$ then d is safe.*

The proof is standard [23, 27, 29] and is deferred to the extended report [28].

As remarked in Section 2, for relative completeness, it suffices to limit the position of extra parameters (i.e., universal quantifiers) to one before each function-type parameter. To this end, we define $\mathit{ushape}(\sigma)$ to be a simple type such that, for a function of the simple type $\mathit{ushape}(\sigma)$, σ has just the sufficient extra parameters. Formally, $\mathit{ushape}(\sigma)$ is defined by the rules below.

$$\frac{}{\mathit{ushape}(\{u \mid \theta\}) = \texttt{int}} \qquad \frac{}{\mathit{ushape}(\star) = \star}$$

$$\frac{\mathit{tshape}(\sigma) = \texttt{int}}{\mathit{ushape}(x{:}\sigma \to \tau) = \mathit{ushape}(\sigma) \to \mathit{ushape}(\tau)}$$

$$\frac{\mathit{tshape}(\sigma') = \texttt{int} \quad \mathit{tshape}(\sigma) \in \to}{\mathit{ushape}(\forall y{:}\sigma'.x{:}\sigma \to \tau) = \mathit{ushape}(\sigma) \to \mathit{ushape}(\tau)}$$

Note that $\mathit{ushape}(\sigma)$ is defined only for σ that has one extra parameter just before a function-type parameter. Also, for σ such that $\mathit{ushape}(\sigma)$ is defined, we have $\mathit{ushape}(\sigma) = \mathit{tshape}(\sigma)$. We are now ready to state the relative completeness theorem, which says that if a program is safe then it can be typed with extra parameter additions (even when their positions are restricted to the pattern above).

THEOREM 3.7 (Relative Completeness). *If d is safe, then there exists Δ such that $\Delta \vdash_\forall d$ and $\mathit{ushape}(\Delta(F)) = \mathit{sty}(F)$ for each $F\ \overrightarrow{x}\ \texttt{=}\ e \in d$.*

We defer the proof to the extended report [28]. The proof adopts the ideas from the work on relatively complete program logics for higher-order procedural languages [6–8] that instantiate the extra parameters by the base-type encoding of the function closures. It is worth noting that, as a corollary of Theorem 3.7, it follows that the refinement type system is relatively complete for first-order programs (i.e., programs without function-type parameters) even without the extra parameter extension.

EXAMPLE 3.8. We show how $\vdash_\forall$ types d_2 and d_3 from Example 3.4. First, we have $\Delta_2 \vdash_\forall d_2$ where

$$\Delta_2(\texttt{app}) = \forall a{:}\texttt{int}.f{:}(\{u \mid u \geq a\} \to \star) \to x{:}\{u \mid u \geq a\} \to \star$$
$$\Delta_2(\texttt{check}) = x{:}\texttt{int} \to y{:}\{u \mid u \geq x\} \to \star$$
$$\Delta_2(\texttt{main}) = i{:}\texttt{int} \to \star$$

Note that the types correspond to the ones used to type Figure 2 in Section 2.1. Also, note that *ushape*(Δ_2(app)) = *sty*(app), *ushape*(Δ_2(check)) = *sty*(check), and *ushape*(Δ_2(main)) = *sty*(main). It is easy to see that check can be given the type Δ_2(check). We show that app can be given the type Δ_2(app). We type the body of app under the type environment $\Delta_2, a:\mathtt{int}, f:\{u \mid u \geq a\} \rightarrow \star, x:\{u \mid u \geq a\}$. Therefore, the application $f\ x$ type-checks to give the type $\star$. To type the other branch, app $f\ (x+1)$, we instantiate Δ_2(app) with a at app to give the type

$$f{:}(\{u \mid u \geq a\} \rightarrow \star) \rightarrow x{:}\{u \mid u \geq a\} \rightarrow \star$$

The type is then used to give the application app f the type $x:\{u \mid u \geq a\} \rightarrow \star$, which in turn gives the branch the type $\star$. Finally, main can be given the type Δ_2(main) by instantiating Δ_2(app) with i.

Likewise, let Δ_3 be the following type environment.

$$\begin{array}{l}\Delta_3(\mathtt{app3}) = \forall a{:}\mathtt{int}.f{:}\tau_1 \rightarrow \forall b{:}\mathtt{int}.g{:}\tau_2 \rightarrow \star\\ \Delta_3(\mathtt{app}) = x{:}\mathtt{int} \rightarrow \forall a{:}\mathtt{int}.f{:}(\{u \mid u \geq x\} \rightarrow \star) \rightarrow \star\\ \Delta_3(\mathtt{succ}) = \forall b{:}\mathtt{int}.f{:}(\sigma \rightarrow \star) \rightarrow x{:}\sigma \rightarrow \star\\ \Delta_3(\mathtt{check}) = x{:}\mathtt{int} \rightarrow y{:}\{u \mid u \geq x\} \rightarrow \star\\ \Delta_3(\mathtt{main}) = i{:}\mathtt{int} \rightarrow \star\end{array}$$

such that $\tau_1 = \{u \mid u \geq a\} \rightarrow \star$, $\tau_2 = \forall c:\mathtt{int}.\tau_1 \rightarrow \star$, and $\sigma = \{u \mid u \geq b\}$. Note that *ushape*($F$) = *sty*($\Delta_3(F)$) for each F = app3, app, succ, check, and main. We show that $\Delta_3 \vdash_\forall d_3$ by following the same instantiation scheme used to type Figure 3 in Section 2.1. We type the then branch of app3 by instantiating app3 and succ with a, and instantiating the resulting app3 (succ f) with a (or with an arbitrary side-effect-free integer-type expression, because b does not appear free in its scope in Δ_3(app3)). The else branch can be typed by instantiating g with an arbitrary side-effect-free integer-type expression (because c does not appear free in its scope in Δ_3(app3)). Then, to type main, we instantiate app3 with i, and instantiate app3 (check i) with an arbitrary side-effect-free integer-type expression.

The examples show that instantiating via simple expressions are often sufficient for verifying safety, and not all quantifiers may even be needed. (Contrast this with the instantiation via Gödel numbering used in the proof of Theorem 3.7.[9]) We take advantage of the observation in the type inference method described in Section 4.

4. Type Inference

The type inference framework is based on, and extends the recent work on the counterexample-guided approach to refinement type inference [18, 26] (but, the idea may be adopted to extend the back-end of other refinement type inference systems like [13, 27]). To allow smooth adoption and use the existing type inference algorithms mostly as a blackbox, we implement the extra parameter addition as a program translation as in Section 2, instead of modifying the underlying type system to model it by universally quantified types as in Section 3.2.

Figure 10 shows the overview of the type inference process. In Step 1, we translate the given program by adding extra parameters as done in Section 2. Here, we maintain and use a *parameter substitution* for instantiating the extra parameters. We initialize the instantiation expressions to arbitrary constants (e.g., 0), and then perform a counterexample-guided refinement type inference over the translated program.

Following the counterexample-guided abstraction refinement (CEGAR) scheme popularized in model checking [3, 12], in the counterexample-guided refinement type inference, we maintain a

[9] This can be seen as a difference between just checking safety and expressing the precise semantics as weakest preconditions.

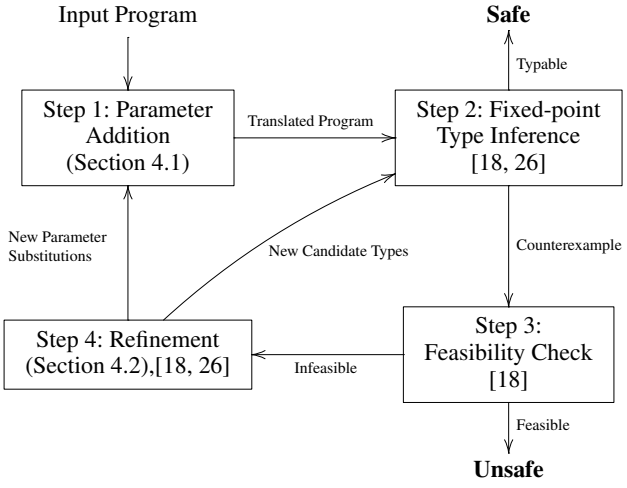

Figure 10. The type inference overview.

set of *candidate* refinement types (or the refinement predicates that comprise them). Then, we try to find a type assignment for the program within the candidates via the standard fixed-point type inference process (Step 2 in Figure 10). The fixed-point type inference uses an automated FOL theorem prover, such as a SMT solver, to decide the FOL implications discharged at the subtyping rule **Sb**. We refer interested readers to the previous work [18, 26] for details.

A *counterexample* is generated when the candidates are found insufficient, and we analyze the reason for the failure to either validate the counterexample (Step 3), or update the candidate types or the parameter substitution to eliminate the spurious counterexample (Step 4). In the refinement step, the previous work [18, 26] only generated new candidate types (i.e., the arrow from Step 4 to Step 2). But, because of the incompleteness of the underlying type system, some spurious counterexamples are impossible to refute by just adding new candidate types. *The key component of our new type inference method is generating new parameter substitutions in such a situation* (i.e., the arrow from Step 4 to Step 1).

Then, the CEGAR process is repeated with the updated candidate types or the updated parameter substitutions. The former case is identical to the previous work, and in the latter case, we repeat from Step 1 to re-translate the program with the updated parameter instantiations.

4.1 Adding Extra Parameters

We add an extra parameter just before each function-type parameter. Also, we assign a unique label to each sub-expression of the program where a parameter instantiation occurs. (Note that such places are syntactically determined.) Let L be the set of the labels and *bvs*(ℓ) denote the base-type variables that can occur free in the scope of the sub-expression with the label $\ell \in L$.

The type inference process maintains the *parameter substitution* P that maps each ℓ to a side-effect-free integer-type expression o such that *fv*(o) $\subseteq$ *bvs*(ℓ). We use $P(\ell)$ as the instantiation expression at ℓ. We restrict the range of P, and therefore, the instantiation expressions, to linear arithmetic expressions. That is, $P(\ell)$ is restricted to be of the form

$$c_0 + c_1x_1 + \cdots + c_nx_n$$

where *bvs*(ℓ) = $\{x_1, \ldots, x_n\}$ and $c_0, \ldots, c_n$ are integer constants. This is an important design choice motivated both by the desire to minimize the complexity of the type inference, and also by the ob-

servation that sufficient instantiations tend to be simple expressions in practice (cf. Example 3.8).

The parameter substitution P may be initialized arbitrarily, for example, $P(\ell) = 0$ for all ℓ. We translate the input program by adding the extra parameters to the function definitions and replacing each instantiation site e with the application $e\,P(\ell)$ where ℓ is the label of e.

EXAMPLE 4.1. Recall the program d_2 in Example 3.4. We add an extra parameter a just before the function-type parameter f of `app`. Let ℓ_1 and ℓ_2 be the labels of the occurrences of `app` in the body of `app` and `main` respectively. We have $L = \{\ell_1, \ell_2\}$, $\mathit{bvs}(\ell_1) = \{a, x\}$, and $\mathit{bvs}(\ell_2) = \{i\}$. Suppose that the parameter substitution is $P = \{\ell_1 \mapsto a, \ell_2 \mapsto i\}$. Then, d_2 is translated to the program d_P shown below.

$$\begin{array}{l}
\texttt{app}\ a\ f\ x = \texttt{if}\ *\ \texttt{then}\ \texttt{app}\ a\ f\ (x+1)\ \texttt{else}\ f\ x\\
\texttt{check}\ x\ y = \texttt{if}\ *\ \texttt{then}\ \texttt{assume}\ (x \le y); \texttt{check}\ x\ y\\
\qquad\qquad\ \texttt{else}\ \texttt{assume}\ \neg(x \le y); \texttt{assert}\ 0; \texttt{check}\ x\ y\\
\texttt{main}\ i = \texttt{app}\ i\ (\texttt{check}\ i)\ i
\end{array}$$

4.2 Counterexample-Guided Refinement

In MoCHi [18], a counterexample is a *straightline higher-order program* (SHP) that is untypable with the current candidate types. SHP is defined to be a recursion-free slice of the target program obtained by copying functions and removing branches so that it contains no branches and every function occurrence is "linear" (i.e., each function is called only once).[10] Intuitively, a counterexample corresponds to the (abstract) program path taken to reach the assertion failure.

We check if the counterexample SHP is feasible. This part is identical to the previous work [18] and is done by symbolically evaluating the SHP. If it is feasible, then the program is determined unsafe and we are done. Otherwise, we attempt to infer refinement types for the SHP. If the SHP is found typable, then the inferred refinement types (or the refinement predicates embedded therein) are added to the candidates and the CEGAR iteration returns to the fixed-point type inference phase (i.e., Step 2 of Figure 10).

We infer types for counterexamples by using the techniques from the previous work [18, 26, 27] that reduce the inference problem to solving constraints over FOL formulas and *predicate variables* such that the predicate variables serve as placeholders of the refinement predicates to be inferred. We refer to the previous work for details.

EXAMPLE 4.2. Given d_P from Example 4.1, MoCHi may generate the SHP d_S shown below.

$$\begin{array}{l}
\texttt{app1}\ a\ f\ x = \texttt{app2}\ a\ f\ (x+1)\\
\texttt{app2}\ a\ f\ x = f\ x\\
\texttt{check}\ x\ y = \texttt{assume}\ \neg(x \le y); \texttt{assert}\ 0\\
\texttt{main}\ i = \texttt{app1}\ i\ (\texttt{check}\ i)\ i
\end{array}$$

Note that d_S is recursion-free and linear. The SHP is typable and MoCHi may infer refinement types Δ such that

$$\begin{array}{l}
\Delta(\texttt{app1}) = \Delta(\texttt{app2}) =\\
\quad a{:}\texttt{int} \to f{:}(\{u \mid u \ge a\} \to \star) \to x{:}\{u \mid u \ge a\} \to \star\\
\Delta(\texttt{check}) = x{:}\texttt{int} \to y{:}\{u \mid u \ge x\} \to \star\\
\Delta(\texttt{main}) = i{:}\texttt{int} \to \star
\end{array}$$

Note that $\Delta \vdash d_S$.

As remarked above, because of the incompleteness of the underlying type system (i.e., $\vdash$), sometimes, a counterexample that is detected to be infeasible is also found untypable. (The feasibility check is actually relatively complete.) In such a situation, we infer a new parameter substitution P_R for the SHP so that the SHP with each of its instantiation site ℓ instantiated with $P_R(\ell)$ is typable. Then, we update the parameter substitution with P_R, that is, we set $P := P_R \cup P|_{L \setminus L_{\text{SHP}}}$ where L_{SHP} is the labels of the SHP. And, the CEGAR process returns to the parameter addition phase to re-translate the program (i.e., Step 1 of Figure 10).

The parameter substitution inference proceeds as follows. First, we assign labels to the SHP such that every copy of a subexpression gets the same label as the one in the original. (Note that because of function copying, the same expression in the original program can have multiple copies in the SHP.) Next, we introduce a *parameter substitution template*, P_T, that maps each $\ell \in L_{\text{SHP}}$ to a linear arithmetic expression template $p_0 + p_1 x_1 + \cdots + p_n x_n$, where $\mathit{bvs}(\ell) = \{x_1, \ldots, x_n\}$ and $p_0, \ldots, p_n$ are fresh integer variables. This, in turn, induces a SHP *template* that has $P_T(\ell)$ as the instantiation expression at each $\ell \in L_{\text{SHP}}$.

Now, the problem of inferring P_R is reduced to that of inferring an appropriate integer substitution for the integer variables $\overrightarrow{p}$ in the template. We reduce the problem to constraint solving by generating a FOL constraint over $\overrightarrow{p}$ of the form $\forall\overrightarrow{x}.\theta$ (for some non-linear FOL formula θ over the variables $\overrightarrow{p}$ and $\overrightarrow{x}$). That is, the SHP is typable with the instantiation expression $P_T(\ell)\rho$ at each ℓ iff $(\forall\overrightarrow{x}.\theta)\rho$ holds where ρ is a substitution that maps $\overrightarrow{p}$ to integer constants. Although the problem of solving quantified integer non-linear FOL constraints is undecidable in general, the recently proposed constraint solving techniques based on Farkas' lemma [4, 9, 24] have shown effective for our application in many cases. We describe the constraint generation process in Section 4.2.1 and the constraint solving process in Section 4.2.2.

4.2.1 Constraint Generation

The constraint generation algorithm is essentially the same as the ones from the previous work [18, 26, 27] used for candidate type inference. We give a brief review of the algorithm. For each function in the program, we prepare type templates containing predicate variables that serve as placeholders of the refinement predicates to be inferred. Then, we generate constraints over the predicate variables in the standard way. That is, we apply the typing rules to the SHP but restricting the application of the subsumption rule **Sub** to just the argument position of function applications (cf. Section 3.1). This generates constraints over the type templates, which, in turn, reduce to Horn-clause-like constraints over the predicate variables. (See the extended report [28] for the formal definition of the constraint generation rules.) Because SHP is recursion-free and linear, the generated Horn clauses are non-recursive, and we can obtain an equivalent FOL formula.

EXAMPLE 4.3. Consider the SHP d'_S shown below, which is equivalent to d_S from Example 4.2 except that the extra parameter of `app1` is instantiated by 0 instead of i.

$$\begin{array}{l}
\texttt{app1}\ a\ f\ x = \texttt{app2}\ a\ f\ (x+1)\\
\texttt{app2}\ a\ f\ x = f\ x\\
\texttt{check}\ x\ y = \texttt{assume}\ \neg(x \le y); \texttt{assert}\ 0\\
\texttt{main}\ i = \texttt{app1}\ 0\ (\texttt{check}\ i)\ i
\end{array}$$

It is easy to see that the counterexample d'_S is infeasible (i.e., is safe), and MoCHi is also able to detect the infeasibility. However, d'_S is untypable with the underlying type system $\vdash$ because of the inappropriate instantiation expression.

To infer a new instantiation expression, we prepare the parameter substitution template P_T for the instantiation sites of d'_S, and

[10] [26] defines a counterexample to be simply a currently-untypable recursion-free program slice (i.e., possibly containing branches and non-linearity), but uses linear intersection types to obtain a similar effect.

from it, obtain the SHP template d_T shown below.

$$\begin{array}{l}\texttt{app1}\ a\ f\ x = \texttt{app2}\ (p_0 + p_1 a + p_2 x)\ f\ (x+1)\\ \texttt{app2}\ a\ f\ x = f\ x\\ \texttt{check}\ x\ y = \texttt{assume}\ \neg(x \le y); \texttt{assert}\ 0\\ \texttt{main}\ i = \texttt{app1}\ (p_3 + p_4 i)\ (\texttt{check}\ i)\ i\end{array}$$

where $p_0, \ldots, p_4$ are free integer variables. Next, we generate the constraints by preparing the type template Δ such that

$$\begin{array}{l}\Delta(\texttt{app1}) =\\ a{:}\{u \mid P_1(u)\} \to (\{u \mid P_2(a,u)\} \to \star) \to \{u \mid P_3(a,u)\} \to \star\\ \Delta(\texttt{app2}) =\\ a{:}\{u \mid P_4(u)\} \to (\{u \mid P_5(a,u)\} \to \star) \to \{u \mid P_6(a,u)\} \to \star\\ \Delta(\texttt{check}) = x{:}\{u \mid P_7(u)\} \to y{:}\{u \mid P_8(x,u)\} \to \star\\ \Delta(\texttt{main}) = i{:}\texttt{int} \to \star\end{array}$$

This generates the following non-recursive set of Horn-clause-like constraints on the predicate variables $P_1, \ldots, P_8$.

$$\begin{array}{ll}\forall a, x. P_1(a) \wedge P_3(a,x) \Rightarrow P_4(p_0 + p_1 a + p_2 x) & \\ \forall a, x, u. P_1(a) \wedge P_3(a,x) \wedge P_5(p_0 + p_1 a + p_2 x, u) \Rightarrow P_2(a,u) & \\ \forall a, x. P_1(a) \wedge P_3(a,x) \Rightarrow P_6(p_0 + p_1 a + p_2 x, x+1) & \\ \forall a, x, u. P_4(a) \wedge P_6(a,x) \Rightarrow P_5(a,x) & \\ \forall x, y. P_7(x) \wedge P_8(x,y) \Rightarrow x \le y & \\ \forall i. \top \Rightarrow P_1(p_3 + p_4 i) & \forall i. \top \Rightarrow P_7(i)\\ \forall i, u. P_2(p_3 + p_4 i, u) \Rightarrow P_8(i,u) & \forall i. \top \Rightarrow P_3(p_3 + p_4 i, i)\end{array}$$

Systematically simplifying the constraints by computing the least solutions for the predicate variables in a bottom-up manner, we obtain an equisatisfiable FOL formula on the variables $p_0, \ldots, p_4$ shown below.

$$\begin{array}{l}\forall x, y, z.\\ \quad p_4 x = p_4 z \wedge p_1 p_4 x + p_2 z = (p_1 p_4 + p_2)(y-1) \Rightarrow x \le y\end{array}$$

4.2.2 Constraint Solving

The constraint generation process above returns a constraint of the form $\forall \overrightarrow{x}.\theta$, where θ is a quantifier-free non-linear FOL formula over the variables $\overrightarrow{p}$ and $\overrightarrow{x}$. (More precisely, θ is linear over $\overrightarrow{x}$ with coefficients over $\overrightarrow{p}$.)[11] The goal of the constraint solving phase is to find an assignment ρ for $\overrightarrow{p}$ satisfying $\forall \overrightarrow{x}.\theta$.

We solve the constraints by adopting Gulwani et al.'s approach [9]. We give a brief overview of the idea. First, we use Farkas' lemma to remove the universal quantifications in the constraint $\forall \overrightarrow{x}.\theta$, and obtain a constraint of the form $\exists \overrightarrow{r}.\phi$, where ϕ is a quantifier-free non-linear FOL formula on the variables $\overrightarrow{p}$ and $\overrightarrow{r}$. Then, we translate the constraint to a SAT formula by modeling integer variables as bit-vectors and integer operations such as addition, multiplication, and comparison as boolean operations. Finally, we use a state-of-the-art SAT solver to find a substitution ρ for $\overrightarrow{p}$ and $\overrightarrow{r}$ that satisfies ϕ.

We describe the approach in a more detail. Given a universally quantified constraint $\forall \overrightarrow{x}.\theta$, we convert θ to an equivalent formula of the form $\bigwedge_i \theta_i$, where each θ_i is of the form $\neg(\bigwedge_{j=1,\ldots,m_i} e_{i,j} \ge 0)$ such that each $e_{i,j} \ge 0$ is a linear inequality on $\overrightarrow{x}$ with polynomials on $\overrightarrow{p}$ as coefficients. We translate each θ_i by applying Farkas' lemma.

THEOREM 4.4 (Farkas' lemma). *Consider the following system of inequalities over real-valued variables $x_1, \ldots, x_n$.*

$$\begin{bmatrix} c_{1,0} + c_{1,1}x_1 + \ldots + c_{1,n}x_n \ge 0 \\ \vdots \qquad\quad \vdots \qquad\qquad \vdots \\ c_{m,0} + c_{m,1}x_1 + \ldots + c_{m,n}x_n \ge 0 \end{bmatrix}$$

The system is unsatisfiable iff there exist non-negative reals r_0, r_1, …, r_m such that

- $r_0 + r_1 c_{1,0} + \cdots + r_m c_{m_0} = -1$, *and*
- $r_1 c_{1,j} + \cdots + r_m c_{m_j} = 0$ *for each* $j = 1, \ldots, n$.

Farkas' lemma is incomplete for integers because the "only if" direction does not always hold. (Trivially, the "if" direction holds even for integers.) The incompleteness, however, has not affected the experiments in Section 5. (See Section 4.3 for the list of limitations with our approach.)

EXAMPLE 4.5. Recall the constraints generated from the SHP template d_T in Example 4.3. We translate the constraints to the equivalent form below.

$$\forall x, y, z.\ \neg \begin{pmatrix} p_4 x & & -p_4 z \ge 0 \wedge \\ -p_4 x & & +p_4 z \ge 0 \wedge \\ p_1 p_4 + p_2 & +p_1 p_4 x\ -(p_1 p_4 + p_2) y & +p_2 z \ge 0 \wedge \\ -p_1 p_4 - p_2 & -p_1 p_4 x\ +(p_1 p_4 + p_2) y & -p_2 z \ge 0 \wedge \\ -1 & +x\ -y & \ge 0 \end{pmatrix}$$

Applying Farkas' lemma, the constraint is further translated to the following form.

$$\begin{array}{l}\exists r_0, r_1, r_2, r_3, r_4 \ge 0.\\ \quad r_0 + (p_1 p_4 + p_2)(r_3 - r_4) - r_5 = -1 \wedge\\ \quad p_4(r_1 - r_2) + p_1 p_4 (r_3 - r_4) + r_5 = 0 \wedge\\ \quad -(p_1 p_4 + p_2)(r_3 - r_4) - r_5 = 0 \wedge\\ \quad -p_4(r_1 - r_2) + p_2(r_3 - r_4) = 0\end{array}$$

We reduce the constraints to SAT by modeling integers as bit-vectors, and apply SAT solving to find a satisfying assignment. A possible solution is

$$\begin{array}{l} p_0 = 0,\ p_1 = 1,\ p_2 = 0,\ p_3 = 0,\ p_4 = 1\\ r_0 = 1,\ r_1 = 0,\ r_2 = 0,\ r_3 = 0,\ r_4 = 1,\ r_5 = 1\end{array}$$

And from the solution, we obtain the substitution

$$\rho = \{p_0 \mapsto 0, p_1 \mapsto 1, p_2 \mapsto 0, p_3 \mapsto 0, p_4 \mapsto 1\}$$

Note that applying the inferred parameter substitution ρ to d_T results in the program equivalent to d_S from Example 4.2. Finally, we update the parameter substitution P via $P_T\rho$ to translate the input program, which results in the program d_P from Example 4.1, and is typable by the underlying type system. While the example only require a variable as the instantiation expression, this is not the case in general (even when the instantiations are restricted to linear arithmetic expressions). Section 5 shows examples needing non-variable instantiation expressions.

We use the iterative approach of Gulwani et al. [9] to model integers as bit-vectors: we start with a low number of bits, and gradually increase the number of bits until a solution is found. Because of the incompleteness limitation, it is possible that a solution is not found even though the process is given an infeasible SHP. (cf. Section 4.3.) Section 5 shows heuristics for coping with the issue.

New candidate types with new parameter substitutions It is often desirable to infer new candidate types along with the new parameter substitutions. For instance, in Example 4.5, rather than returning to Step 1 of Figure 10 after the updated parameter substitution $P_T\rho$ is inferred, it is better to infer candidate types for the translated counterexample SHP $d_T\rho$ so that the counterexample is immediately eliminated from the future CEGAR iterations.

Inferring parameter substitution for prior counterexamples For simplicity, we have shown how to infer a parameter substitution for one counterexample SHP. In practice, it is better to record the counterexamples that have been encountered (or, the constraints

[11] We assume that constant types only embed quantifier-free linear arithmetic refinement predicates.

generated from them), so that the inferred parameter substitution is able to also refute the past counterexamples. This can be done by a minor modification to the algorithm because constraints from multiple counterexamples simply add up as conjunctions. We take this approach in the implementation described in Section 5.

4.3 Limitations

Our type inference algorithm is incomplete as it is not able to decide the typability of all programs. This is expected because the underlying problem (i.e., safety verification) is undecidable. Here, we list the source of incompleteness.

(1) Like the predicate abstraction in first-order software model checking [12], we restrict the refinement predicates to an efficiently decidable first-order theory, such as the quantifier-free theory of linear arithmetic. Consequently, there are programs, even ones that are first-order, that we cannot verify. Also, predicate discovery uses various heuristics such as interpolation, and is in general incomplete even for the theory subset (but, there are approaches to make this part complete [14]).

(2) As remarked in Section 4.1, we restrict instantiation expressions to linear arithmetic expressions. While this has been sufficient for many cases, as we show in Section 5, there are programs that we cannot verify because of the restriction.[12] (But, as we show there, simple heuristics can often be used to handle such cases.)

(3) The parameter substitution inference is invoked only when the counterexample is untypable with the current substitutions, because otherwise it may simply infer the same substitutions. For some programs, the parameter substitution inference becomes forever blocked because the program generates infinitely many counterexamples each of which can be refuted by just updating the candidate types.[13] (See Section 5 for an example.)

(4) As remarked in Section 4.2.2, Farkas' lemma is incomplete for integers.

5. Implementation and Experiments

We have implemented a prototype of the type inference algorithm as an extension to the higher-order model checker MoCHi [18]. It takes (direct-style and CPS) OCaml programs. We use CVC3 [2] for the SAT-based non-linear constraint solving described in Section 4.2.2. A web interface of the implementation and the benchmark programs from the experiments are available online [28].

We ran experiments on small but representative higher-order programs. Table 1 summarizes the results. The column S is the size measured in the number of words in the program. O is the largest order of the functions in the program. (Base-type values are order-0, and order-n functions only take arguments of order less than n.) The column $|\overrightarrow{p}|$ is the total number of the variables $\overrightarrow{p}$ that occur in the range of the parameter substitution template P_T (cf. Section 4.2). #E is the number of extra parameters added before each function-type parameter. (#E > 1 is used in the heuristics described below.) #I is the number of CEGAR iterations. The column T is the running time in seconds, and P/T is the fraction of the running time spent on the parameter substitution inference. NV marks the programs that required non-variable instantiation expressions. The experiments were conducted on a machine with Intel Xeon E5620 2.4GHz CPU and 4GB RAM.

Note that the programs are higher-order (i.e., O > 1). The programs are safe, but none of them can be verified by the previous refinement type systems proposed for automated higher-order program verification [13, 18, 23, 26, 27]. We describe each program.

program	S	O	$\lvert\overrightarrow{p}\rvert$	#E	#I	T	P/T	NV
d_2	40	2	3	1	2	0.43s	40%	
d_3	56	3	10	1	3	1.76s	68%	
fhnhn	26	2	2	1	1	0.15s	12%	
repeat-add	51	2	5	1	3	1.05s	7%	
app-leq	29	2	2	1	1	0.26s	41%	✓
app-lin-ord2	37	2	2	1	1	0.22s	38%	✓
app-lin-ord3	45	3	6	1	1	0.96s	84%	✓
app-succ	50	2	5	1	2	0.36s	37%	
app-succ0	50	2	5	–	–	—	—%	
a-test-upd	80	2	9	2	7	2.06s	32%	
a-checksum	76	2	6	2	8	3.38s	29%	
a-max	78	2	8	1	4	7.32s	26%	
l-forall-leq	69	2	8	1	2	2.68s	51%	
l-len-append	126	2	22	1	2	0.73s	35%	✓
l-isort	170	2	39	–	–	—	—%	

Table 1. Experiment results.

- d_2 and d_3 are from Example 3.4.
- fhnhn compares the return value of two function arguments:

```
let f x y = assert (x () = y ()) in
let h x () = x in let main n = f (h n) (h n)
```

- repeat-add makes the closure add n for some n ≥ 0, applies it to 0 for k-times for some k > 0, and asserts that the result is not less than n:

```
let add x1 x2 = x1 + x2 in
let rec repeat f k x = if k <= 0 then x
              else f (repeat f (k - 1) x) in
let main n k =
  if n >= 0 && k > 0 then
    assert (repeat (add n) k 0 >= n)
```

- app-leq (resp. app-lin-ord2) is the program below with e $\equiv$ (a<=b) (resp. e $\equiv$ (4*a+2*b)).

```
let app f x = f x in
let check x y = assert (x = y) in
let main a b = app (check e) e
```

- app-lin-ord3 is app-lin-ord2 "lifted" one order higher.
- app-succ calls check i i for some non-deterministically-chosen $i \geq$ n:

```
let succ f x = f (x + 1) in
let rec app f x =
  if * then app (succ f) (x - 1) else f x in
let check x y = assert (x = y) in
let main n = app (check n) n
```

- app-succ0 is app-succ with n in the body of main fixed to 0.
- In a-xxx, we model and check array manipulating programs. We model an array as the pair consisting of its size and the function that maps indices to the array elements:

```
let make_array n =
  (n, fun i -> assert (0 <= i && i < n); 0)
let upd (n, ar) i x =
  assert (0 <= i && i < n);
  (n, fun j -> if j = i then x else ar j)
```

[12] The limitation is also linked with (1) above, because allowing more complex instantiations has no benefit when the underlying theorem prover cannot decide formulas embedding them.

[13] This is an instance of a more general issue with the parameter substitution inference not inferring sufficiently general substitutions.

The array benchmarks extend the ones from the previous work [18] that demonstrate the application of higher-order program verification to the verification of data structure properties.

- `a-test-upd` tests array updates. It creates an array of size `n` and updates the `i`-th element to `x` for some `i` $\in \{0, \ldots, n-1\}$. The program then reads the element and checks that the result is equal to `x` as expected:

```
let test (n, ar) i x = assert (ar i = x) in
let main n i x =
  if 0 <= i && i < n then
    test (upd (make_array n) i x) i x
```

  Note that the program also checks for array bounds violation.

- `a-checksum` creates an array of size 2, updates the first and the second elements to `a` and `b` respectively, and asserts that the sum of the two array elements equals `a` + `b`:

```
let checksum (n, ar) x =
  assert ((ar 0) + (ar 1) = x) in
let main a b =
  checksum
   (upd (upd (make_array 2) 0 a) 1 b) (a + b)
```

- `a-max` creates an array of size `n` whose elements are from the set $\{0, \ldots, x\}$ where `x` > 0, computes the maximum element `m` of the array, and asserts that `m` $\leq$ `x`.

- The programs `l-xxx` model and check list operations. We model a list by a pair of its length and a function from a non-negative integer i to the i-th element. We model the core list operations as follows:

```
let nil = (0, fun i -> assert false)
let cons a (len, l) =
  (len + 1, fun i -> if i=0 then a else l (i-1))
let hd (len, l) = l 0
let tl (len, l) = (len - 1, fun i -> l (i + 1))
let is_nil (len, l) = len=0
```

  - `l-forall-leq` creates a list of the length `n` whose `i`-th element is `n` $-$ `i`, and asserts that `x` $\leq$ `n` holds for all element `x` of the list.
  - `l-len-append` appends two lists of the length `len1` and `len2`, computes the length of the resulting list, and asserts that it is not greater than `len1` + `len2`.
  - `l-isort` creates a list, sorts the list via an insertion sort, and then checks that the result is actually sorted.

The implementation was able to successfully verify the programs by automatically inferring appropriate extra parameter substitutions, except for `app-succ0` and `l-isort`. Before we discuss the reason for failure on these two, we describe the heuristic that was needed for verifying `a-test-upd` and `a-checksum`.

Recall that we limit instantiation expressions to linear arithmetic expressions. With this restriction, it can be shown that `a-test-upd` and `a-checksum` (and also `l-isort`) are actually untypable when we are allowed to add only one extra parameter before a function-type parameter.[14] However, `a-test-upd` and `a-checksum` can be typed if we are allowed to add two extra parameters before a function-type parameter, even with the linear-instantiation-expression restriction. (Note that #E is 2 for these programs in Table 1.)

For instance, in `a-checksum`, to show the safety of the call to `checksum`, we need to infer the property that the second argument equals the sum of the first and the second elements of the function-encoded array passed as the first argument. Because the two elements of the array are independent (i.e., the arguments `a` and `b` of `main`), it is not possible to express such a fact via a refinement type with just one (linearly instantiatable) extra parameter. Nonetheless, the property can be expressed even under the linear-instantiation-expression restriction by using two extra parameters. For example, the following type is sufficient:

$$\forall a{:}\texttt{int}.\forall b{:}\texttt{int}.\\ (\texttt{int} \times (x{:}\texttt{int} \to \{u \mid \theta\})) \to \{u \mid u = a + b\} \to \star$$

where $\theta \equiv (x = 0 \Rightarrow u = a) \wedge (x = 1 \Rightarrow u = b)$, and $\times$ is the pair type constructor. We use a simple heuristic to progressively increase the number of extra parameters that are added before each function-type parameter: we increment the number by one when the parameter substitution inference fails to find a solution to the constraints, with some pre-defined threshold on the number of bits used in the bit-vector modeling (cf. Section 4.2.2). With the heuristic, the implementation was able to verify `a-test-upd` and `a-checksum`. Note that the increased extra parameter heuristic is sound because the soundness theorem (Theorem 3.6) holds for any pattern of extra parameter additions.

The program `l-isort` remains untypable even with the heuristic. In fact, it can be shown that no finite number of extra parameter additions can type the program when the instantiation expressions are restricted to be linear.

The implementation fails to verify `app-succ0` for a different reason. To verify the program, we need to infer the following type for `app` with the extra parameter i.

$$\forall i{:}\texttt{int}.f{:}(\{u \mid u = i\} \to \star) \to x{:}\{u \mid u = i\} \to \star$$

However, each counterexample SHP generated from `app-succ0` is actually typable by assigning (the copies of) `app` the types of the following form whose refinement predicates do not mention the extra parameter:

$$\forall i{:}\texttt{int}.f{:}(\{u \mid u = c\} \to \star) \to x{:}\{u \mid u = c\} \to \star$$

where c is some integer constant. Therefore, the type inference system generates infinitely many counterexample SHPs that are refuted by the types of the above form each time, and the parameter substitution inference is never invoked to infer the necessary type that uses the extra parameter. We leave the issue for future work.

6. Related Work

6.1 Refinement Type Systems

The relatively complete refinement type system of this paper can be seen as DML [30] but with quantifiers (i.e., introduction of Π types) restricted to one before each function-type parameter. Hence, our result shows that DML is actually relatively complete (even when the quantification pattern is restricted). In relation to DML, this paper's contribution is in actually proving relative completeness, and in proposing an automated type inference method.

The previous refinement type systems proposed for automated higher-order program verification [13, 18, 23, 26, 27] lack relative completeness, except for the case when the base-type data domain is finite [16, 17]. We have shown how the type inference systems underlying the verifiers may be extended to attain relative completeness.

6.2 Hoare Logic for Higher-order Languages

There is a long line of research on relatively complete Hoare logic like proof systems for higher-order functional (or procedural) languages. One way to achieve relative completeness is to have a higher-order logic as the *interpretation* logic (i.e., the logic for the

[14] Note that the relative completeness theorem (Theorem 3.7) assumes that arbitrary side-effect-free integer-type expressions can be used for instantiation.

data part) so that the formulas can directly refer to higher-order functions as data (see, e.g., [5, 11, 21, 22]). However, such an approach is difficult to automate as the client verifier must rely on a higher-order logic theorem prover. The dilemma is similar to that arising from explicitly encoding functions as base-type data in the program logic.

The approach of using base-type encoding of functions to achieve relative completeness was first proposed by German et al. [6, 7]. (A similar approach is taken in [8].) Crucial to their approach is the idea of avoiding the explicit encoding in the program logic via quantification, allowing the client verifier to choose appropriate quantifier instantiations. To our knowledge, no actual verifier was built based on their program logic.

The encoding approach is also used in a recent work on relatively complete dependent (refinement) type system [19] where a DML-like type system is extended with linear intersection types, and is shown sound and relatively complete (for terminating programs). Their type system is designed for a type-based complexity analysis, and the paper does not discuss automated verification.

6.3 Quantified First-order Logic Constraint Solving

Our type inference method utilizes an algorithm for deciding the validity a FOL formula of the form $\exists\overrightarrow{x}.\forall\overrightarrow{y}.\theta$ where θ is a quantifier-free arithmetic formula over the variables $\overrightarrow{x}$ and $\overrightarrow{y}$, and if valid, finding appropriate instantiations for $\overrightarrow{x}$ so that the formula can be reduced to $\forall\overrightarrow{y}.\theta[\overrightarrow{e}/\overrightarrow{x}]$.

The problem is studied in the context of *constraint-based* (or, *template-based*) program verification [4, 9, 24] where the existentially quantified variables are used to represent the unknowns in the invariant template. The technique has found wide applicability, including hybrid system verification [20] and program synthesis [25]. Our work adds higher-order program verification to the list.

7. Conclusion

We have presented an automated approach to a relatively complete verification of higher-order functional programs. Our work extends the recent research on refinement type inference for automated program verification. We have extended the underlying refinement type system by adopting the classical result on relatively complete Hoare logic like proof systems for higher-order procedural languages, which shows that certain extra base-type parameter additions are sufficient for relative completeness. Then, we have extended the type inference system by utilizing the techniques from the recent work on quantified FOL constraint solving over template expressions to infer appropriate instantiations for the extra base-type parameters.

Acknowledgments We thank the anonymous reviewers for useful comments. This work is partially supported by Kakenhi 23220001 and 23700026.

References

[1] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, pages 239–250, 1998.

[2] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[4] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, pages 420–432, 2003.

[5] W. Damm and B. Josko. A sound and relatively* complete Hoare-logic for a language with higher type procedures. *Acta Inf.*, 20:59–101, 1983.

[6] S. M. German, E. M. Clarke, and J. Y. Halpern. Reasoning about procedures as parameters. In *Logic of Programs*, pages 206–220, 1983.

[7] S. M. German, E. M. Clarke, and J. Y. Halpern. Reasoning about procedures as parameters in the language L4. *Inf. Comput.*, 83(3):265–359, 1989.

[8] A. Goerdt. A Hoare calculus for functions defined by recursion on higher types. In *Logic of Programs*, pages 106–117, 1985.

[9] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[11] K. Honda, M. Berger, and N. Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *ICALP (2)*, pages 360–371, 2006.

[12] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[13] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, pages 470–485, 2011.

[14] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

[15] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[16] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL*, pages 416–428, 2009.

[17] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188, 2009.

[18] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI*, pages 222–233, 2011.

[19] U. D. Lago and M. Gaboardi. Linear dependent types and relative completeness. In *LICS*, pages 133–142, 2011.

[20] J. Liu, N. Zhan, and H. Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In *EMSOFT*, pages 97–106, 2011.

[21] E.-R. Olderog. Correctness of programs with Pascal-like procedures without global variables. *Theor. Comput. Sci.*, 30:49–90, 1984.

[22] B. Reus and T. Streicher. Relative completeness for logics of functional programs. In *CSL*, pages 470–480, 2011.

[23] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[24] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In *SAS*, pages 53–68, 2004.

[25] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[26] T. Terauchi. Dependent types from counterexamples. In *POPL*, pages 119–130, 2010.

[27] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP*, pages 277–288, 2009.

[28] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs, 2012. `http://www.kb.is.s.u-tokyo.ac.jp/~uhiro/relcomp`.

[29] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[30] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.