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.

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 \( \text{main} \), the program recursively calls \( \text{app} \) non-deterministically to apply the closure check \( i \) to \( i + j \) where \( j \) is the number of times the then branch is taken in \( \text{app} \). 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 \( \text{app} \).

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:

\[
\begin{aligned}
\text{app} &: \forall x \rightarrow f : \{ \{ u : \text{int} \mid u \geq x \} \rightarrow \text{unit} \} \rightarrow \text{unit} \\
\text{check} &: \forall x \rightarrow y : \{ \{ u : \text{int} \mid u \geq x \} \rightarrow \text{unit} \\
\text{main} &: \forall i : \text{int} \rightarrow \text{unit}
\end{aligned}
\]

The type of \( \text{app} \) 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 \( \text{check} \) says that it takes integers \( x \) and \( y \) such that \( x \leq y \). And, the type of \( \text{main} \) 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 \( \text{app} \)’s arguments. Nor, are they able to type and verify the program shown in Figure 3, which uses the function \( \text{success} \) to successively build closures to pass integers larger than \( 1 \) to \( \text{check} \). 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 inexpressible ones. While higher-order program verification is
generally 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
not to let alone inferable ones. While higher-order program verification is
untypable. It is worth noting that the program of Figure 1 does not
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., \( \text{int} = \{u:\text{int} \mid \top\} \).

Therefore, we try to type the body of \( \text{main} \) under the assumption that the
type of \( i \) is \( \{u:\text{int} \mid \top\} \). The most precise type for the
partial application check \( i \) is \( u: \{\text{int} \mid u \geq i\} \to \text{unit} \) (i.e.,
functions that can take any integer at least as large as \( i \)), and the
most precise type for \( i \) is \( \{u:\text{int} \mid u = i\} \) (i.e., integers equal to \( i \)). (Intuitively, \( \{u:B \mid \theta\} \) expresses values of the base type \( B \)
satisfying the refinement predicate \( \theta \), 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 \( \tau \).)

We show that the type systems fail to give the higher-order
function \( \text{app} \) a sufficiently precise type to verify the program’s

\[
f : \{\{u:\text{int} \mid \theta\} \to \text{unit}\} \to x : \{u:\text{int} \mid \varphi\} \to \text{unit}
\]

where the refinement predicates \( \theta \) and \( \varphi \) 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, \( \theta \) and \( \varphi \) are
restricted so that \( \text{fv}(\theta) \subseteq \{u\} \) and \( \text{fv}(\varphi) \subseteq \{f,u\} \),
where \( \text{fv}(\theta) \) denotes the free variables of \( \theta \).)

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

A solution to the set of constraints is \( \theta \equiv \varphi \equiv u \equiv i \),
but this is degenerate because \( \text{main} \)'s variable \( i \) would appear in
app's type. (Recall that \( \theta \), \( \varphi \) 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 \( \theta \), \( \varphi \) such that
\( \text{fv}(\theta) \subseteq \{u\} \) and \( \text{fv}(\varphi) \subseteq \{f,u\} \). 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:

\[
x : \text{int} \to f : \{u: \text{int} \mid u \geq x\} \to \text{unit} \to \text{unit}
\]

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 succ \^\( f \) 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-T 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.

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

where $\tau = \{u: \text{int} \mid u \geq a\}$. 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 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.

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

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

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.

\[
d ::= d \cup \{F \mathrel{\varsubsetneq} e\} \mid \emptyset \\
e ::= x \mid F \mid e \mid \text{let } x = c_1 \text{ in } e_2 \mid e\ x \\
\text{if } * \text{ then } c_1 \text{ else } e_2
\]

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 $i$ = ($app[v2]$ ($check$ $i$))$[i]$ ($app$ $i$)

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. Appendix contains the materials omitted from the main body of the paper.

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, true $= \text{1}$ and false $= \text{0}$.

A program, $d$, is a finite set of function definitions, $F \mathrel{\varsubsetneq} e$, defining a function named $F$ with the formal parameters $\varsubsetneq$ and the body $e$. The notation $\emptyset$ 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 $\text{1}$, $\text{0}$, $\text{1}$, $\text{2}$, 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 let $x = e_1$ 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
0 expression e
expression e
expression e
expression e
expression e
expression e

Figure 5. The reduction rules.

the assume and assertion expressions assume x and assert x, and 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 e0 e1 e2 = (e0 e1) e2. We write e0 e for the series of applications e0 e1 ... en where e = e1, e2, ..., en. We write e1; e2 for let x = e1 in e2 such that x \notin fv(e2), where 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 if x then e1 else e2 can be encoded as

if * then assume x; e1 else let y = ¬x in assume y; e2

which is equivalent for assertion safety.

We define the operational semantics of the language as a step-by-step reduction relation from states to states. A step reduction relation from states to states. A

The reduction relation \( \rightarrow \) is defined by the rules shown in Figure 5. Here, e \( \not\rightarrow \) e' denotes the evaluation of the value returning expression e to the value e', and is defined as F e \( \not\rightarrow \) F e and e \( \not\rightarrow \) e'[\( \lbrack e \rbrack \)(\( \lbrack e \rbrack \)) where \( \lbrack e \rbrack \) is the relation denoting the semantics of a non-assume/assert constant c, so that, for example \( \lbrack i + j \rbrack \) = i + j for integers i and j. (An integer constant is represented by a 0-ary constant.) We let partial applications (i.e., F 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 eval in d and \( \lbrack \cdot \rbrack \) are integer arguments for main (i.e., \( \lbrack \cdot \rbrack \) = \( \lbrack \cdot \rbrack \)). Note that, because of CPS, reductions only occur at the top level. We write e \( \not\rightarrow \) 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 evalin[\( \lbrack d \rbrack \)] \( \not\rightarrow \) fail for any arguments \( \lbrack \cdot \rbrack \) of main.

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

\[
s ::= \ast \mid \text{int} \mid s \rightarrow s'
\]

Here, the type \( \ast \) represents the type of a CPS expression. For each expression e in the program, we write 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 Appendix A.

\[
\tau, \sigma ::= \ast \mid \{u \mid \theta\} \mid \{x : \sigma \rightarrow \tau\}
\]

\[
\begin{align*}
\text{let } x &= e_1 \text{ in } e_2 & \text{where } e_1 \Downarrow e_3 \\
\text{let } x &= \text{assert } li \text{ in } e & \rightarrow e[1/x] \\
\text{let } x &= \text{assume } li \text{ in } e & \rightarrow e[1/x] \\
\text{let } x &= \text{assert } li \text{ in } e & \rightarrow e[1/x] \\
\text{let } x &= \text{assume } li \text{ in } e & \rightarrow e[1/x] \\
\text{if } * \text{ then } e_1 \text{ else } e_2 & \rightarrow e_1 \\
\text{if } * \text{ then } e_1 \text{ else } e_2 & \rightarrow e_2 \\
F e & \rightarrow e'[\lbrack e \rbrack / \lbrack e \rbrack] \quad \text{where } F \lbrack e \rbrack = e' \in d
\end{align*}
\]

Figure 6. The syntax of refinement types.

\[
\begin{align*}
sty(x) &= \text{int} \\
sty(\alpha \times \beta) &= \text{int} \\
sty(\alpha / \beta) &= \text{int} \\
sty(\alpha) &= \text{int} \\
sty(\alpha \rightarrow \beta) &= \text{int} \\
sty(\text{let } x = e \text{ in } e') &= \text{int} \\
sty(\text{if } * \text{ then } e_1 \text{ else } e_2) &= \text{int} \\
sty(\text{let } x = e \text{ in } e') &= \text{int} \\
sty(\text{if } * \text{ then } e_1 \text{ else } e_2) &= \text{int}
\end{align*}
\]

3 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.

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]3).

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 \theta which is a FOL arithmetic formula over the base (i.e., integer) type variables.\footnote{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.} We sometimes abbreviate \{u \mid \theta\} as \text{int} when \theta is a tautology (e.g., \{u \mid \top\} = \text{int}). Intuitively, \{u \mid \theta\} denotes the type of integers u satisfying the formula \theta.

The type \( x : \sigma \rightarrow \tau \) is a dependent function type, consisting of the argument type \sigma and the return type \tau. Intuitively, \( x : \sigma \rightarrow \tau \) 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 \( \sigma \). As usual, \( \rightarrow \) associates to the right.

The type \( \{x \mid \theta\} \) binds \( x \) in \( \theta \). Likewise, \( x : \sigma \rightarrow \tau \) binds \( x \) in \( \tau \) (but not in \( \sigma \)). That is,

\[
\begin{align*}
\text{fv}(x \mid \theta) &= \text{fv}(\theta) \setminus \{x\} \\
\text{fv}(x: \sigma \rightarrow \tau) &= \text{fv}(\sigma) \cup \text{fv}(\tau) \setminus \{x\}
\end{align*}
\]

We sometimes abbreviate \( x : \sigma \rightarrow \tau \) as \( \sigma \rightarrow \tau \) when \( x \) does not occur free in \( \tau \). (It actually suffices to limit \( x \) to occur free in \( \tau \) 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 \( \sigma \), \( \text{tshape}(\sigma) \), is defined as follows:

\[
\begin{align*}
\text{tshape}(\{u \mid \theta\}) &= \text{int} \\
\text{tshape}(x : \sigma \rightarrow \tau) &= \text{tshape}(\sigma) \rightarrow \text{tshape}(\tau)
\end{align*}
\]
\[
\frac{\Gamma \vdash \sigma_2 \leq \sigma_1 \quad \Gamma, x : \sigma_2 \vdash \tau_1 \leq \tau_2}{\Gamma \vdash x : \sigma_1 \rightarrow \tau_1 \leq x : \sigma_2 \rightarrow \tau_2} \quad \text{Sf}
\]

\[
\frac{\Gamma \vdash \star \leq \star}{\Gamma \vdash u \notin f(\Gamma') \quad \left(\left[\Gamma\right] \land \theta_1\right) \Rightarrow \theta_2}{\Gamma \vdash \{u \mid \theta_1\} \leq \{u \mid \theta_2\}} \quad \text{Sb}
\]

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 \( \Gamma \) is a type environment mapping variables and function names to types.

We discuss each typing rule. \textbf{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. \textbf{Vf} types function-type variables and function names by looking up the environment. Here, \( \rightarrow \) denotes the set of simple function types. (We use the meta variable \( \kappa \) to range over both variables and function names.) \textbf{Cst} types constants. Here, \( ty(c) \) denotes the pre-assigned type of the constant \( c \) such that

\( ty(\text{assume}) = x : \text{int} \rightarrow \{u \mid u = x + y\} \)

and \( ty(\text{assert}) = x : \{u \mid u = x + y\} \). Let \( \text{let} \) be self-explanatory. \textbf{App} types applications. Here, \( \tau[x/y] \) is the usual capture-avoiding substitution. If types branch expressions.

\textbf{Sub} is the subsumption rule. The subtyping relation is defined in Figure 8. In \( \text{Sb} \), \( \left[\Gamma\right] \) is the FOL formula denoting the assumptions about the base-type variables in \( \Gamma \), and is defined as follows.

\[
\left[\Gamma\right] = \bigwedge_{v : \left[\sigma\right] \in \Gamma} \theta[v/u]
\]

Like the rule of consequence of the Hoare logic \cite{Hoare69}, \textbf{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 \textit{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.

\textbf{Sub} enforces \textit{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.

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

Let us write \( x: \Delta \rightarrow \tau \) to abbreviate the function type \( x_1 : \sigma_1 \rightarrow \cdots \rightarrow x_n : \sigma_n \rightarrow \tau \) where \( x : \Delta = x_1 : \sigma_1, \ldots, x_n : \sigma_n \). We define the notion of a well-typed program.

\textbf{Definition 3.2} (Well-typed program). We write \( \Delta \vdash d \) if the following conditions hold.

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

2. For each function \( F : \Delta \rightarrow \tau \), we have \( \Delta(F) = x : \Delta \rightarrow \star \).

3. \( \Delta(\text{main}) \) is of the form \( x : \text{int} \rightarrow \star \). (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 \( \Delta \) such that \( \Delta \vdash d \).

The condition (2) says that \( \Delta \) contains fixed-point types for the recursive functions comprising \( d \), and (3) says that \( \text{main} \) is safe to be called with any arguments (recall that \( \text{main} \) 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.

\textbf{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).

\textbf{Example 3.4}. Let the program \( d_1 \) consist of the following functions.

\begin{align*}
\text{app} & = f \star \text{if} \star \text{then app}(x + 1) \star \text{else f x} \star \\
\text{check} & = xy \star \text{if} \star \text{then assume(x \leq y)} \star \text{check x y} \star \\
\text{main} & = \text{app i (check i)}
\end{align*}

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

\begin{align*}
\Delta(\text{app}) & = x : \text{int} \rightarrow f : \{u \mid u \geq x \} \rightarrow \star \rightarrow \\
\Delta(\text{check}) & = x : \text{int} \rightarrow y : \{u \mid u \geq x \} \rightarrow \star \\
\Delta(\text{main}) & = x : \text{int} \rightarrow \star
\end{align*}

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

Next, consider \( d_2 \) shown below.

\begin{align*}
\text{app} & = f \star \text{if} \star \text{then app}(x + 1) \star \text{else f x} \\
\text{check} & = xy \star \text{if} \star \text{then assume(x \leq y)} \star \text{check x y} \star \\
\text{main} & = \text{app (check i) i}
\end{align*}

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 \cite{Narasimhan01, Narasimhan02, Narasimhan03, Narasimhan04}). Note that we cannot simply assign the type \( \Delta(\text{app}) \) from the above but with the order of \( x \) and \( f \) reversed so that

\[
f : \{u \mid u \geq x \} \rightarrow \star \rightarrow x : \text{int} \rightarrow \star
\]

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.

\begin{align*}
\text{app3} & = \text{f g} = \text{if} \star \text{then app3 (succ f) g else g f} \\
\text{app} & = x f = f x \\
\text{succ} & = \text{f (x + 1)} \\
\text{check} & = xy \star \text{if} \star \text{then assume(x \leq y)} \star \text{check x y} \star \\
\text{main} & = \text{app3 (check i) (app i)}
\end{align*}

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
closures $\text{app}^\ell f$ without some non-trivial program reasoning (i.e., $i$ captured in $g$ added the number of times $\text{app}^3$’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 $\text{Sb}$.

Rather than formulating the extra parameter addition as a program translation as done in Section 2, here, we present it as a type quantifier instantiation and the subtyping rule $\text{Sb}$.

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

$$o ::= x \mid e \mid o \circ o$$

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

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

$$x : \text{int} \rightarrow \{ (u | u = 0) \rightarrow * \} \rightarrow *$$

are well-formed refinement types for $F$ such that $s(F) = \text{int} \rightarrow (\text{int} \rightarrow *) \rightarrow *$.

Let $\gamma$ be a disjoint union of variable bindings $x : \sigma$ or universal quantifier bindings $\forall x : \sigma$. The type abbreviation $\gamma \rightarrow \tau$ is defined as follows.

$$\gamma \rightarrow \tau = \begin{cases} \tau & \text{if } \gamma = e \\ \{ \forall x : \sigma. \gamma_1 \} & \text{if } \gamma = \forall x : \sigma. \gamma_1 \\
\{ \forall x : \sigma. \gamma_1 \} & \text{if } \gamma = \forall x : \sigma. \gamma_1 \rightarrow \tau = \tau_1 \\
\{ x : \sigma. \gamma_1 \} & \text{if } \gamma = x : \sigma. \gamma_1 \\
\{ x : \sigma. \gamma_1 \} & \text{if } \gamma = \forall x : \sigma. \gamma_1 \rightarrow \tau = \tau_1 \\
\end{cases}$$

(Note that this subsumes the $\overline{x : \tau} \rightarrow \tau$ abbreviation introduced earlier.) Let $\langle \gamma \rangle$ be the sequence $\gamma$ with each universal quantifier binding $\forall x : \sigma$ replaced with the variable binding $x : \sigma$. That is, $\langle x \rangle = e$, and $\langle x : \sigma, \gamma \rangle = \langle x : \sigma, \gamma \rangle = x : \sigma, \gamma$. We extend the notion of well-typed program to accommodate universal quantifier bindings.

#### Definition 3.5 (Well-typed program – Extended)

We write $\Delta \vdash_\gamma d$ if the following conditions hold.

1. $\Delta$ is a well-formed top-level type environment.
2. For each function $F \overline{\gamma} = e \in d$, we have $\Delta, \langle \overline{\gamma} \rangle \vdash_\gamma e : \ast$ where $\Delta(F) = \overline{\gamma} \rightarrow \ast$.
3. $\Delta[\text{main}]$ is of the form $x : \text{int} \rightarrow \ast$.

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

Note that the only differences from Definition 3.2 are $\langle \gamma \rangle \rightarrow \ast$ and $\langle \overline{\gamma} \rangle$ 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 $\Delta \vdash_\gamma d$ then $d$ is safe.

The proof is standard [23, 27, 29] and is deferred to Appendix D.

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 $\text{ushape}(\sigma)$ to be a simple type such that, for a function of the simple type $\text{ushape}(\sigma)$, $\sigma$ has just the sufficient extra parameters. Formally, $\text{ushape}(\sigma)$ is defined by the rules below.

$$\text{ushape}(\{ u | u = 0 \}) = \text{int} \quad \text{ushape}(\ast) = \ast$$

$$\text{ushape}(\overline{\gamma}) = \text{ushape}(\sigma) \rightarrow \text{ushape}(\tau)$$

$$\text{tshape}(\sigma') = \text{int} \quad \text{tshape}(\sigma) \rightarrow \ast$$

$$\text{ushape}(\forall y : \sigma'. x : \sigma \rightarrow \tau) = \text{ushape}(\sigma) \rightarrow \text{ushape}(\tau)$$

Note that $\text{ushape}(\sigma)$ is defined only for $\sigma$ that has one extra parameter just before a function-type parameter. Also, for $\sigma$ such that $\text{ushape}(\sigma)$ is defined, we have $\text{ushape}(\sigma) = \text{tshape}(\sigma)$. 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 $\Delta$ such that $\Delta \vdash_\gamma d$ and $\text{ushape}(\Delta(F)) = s(F)$ for each $F \overline{\gamma} = e \in d$.

We defer the proof to Appendix E. 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 $\vdash_\gamma$ types $d_2$ and $d_3$ from Example 3.4. First, we have $\Delta_2 \vdash_\gamma d_2$ where

$$\Delta_2(\text{app}) = \{ \forall x : \text{int} : (\{ u | u \geq a \} \rightarrow \ast) \rightarrow x : (\{ u | u \geq a \} \rightarrow \ast) \\ \Delta_2(\text{check}) = x : \text{int} \rightarrow y : (\{ u | u \geq x \} \rightarrow \ast) \\ \Delta_2[\text{main}] = \{ \text{int} \rightarrow \ast \}$$

We extend the typing rules to (non-A-normal-form) side-effect-free integer-type expressions in the obvious way. See Appendix C.
Note that the types correspond to the ones used to type Figure 2 in Section 2.1. Also, note that $\text{ushape}(\Delta_2(\text{app})) = \text{sty}(\text{app})$, $\text{ushape}(\Delta_2(\text{check})) = \text{sty}(\text{check})$, and $\text{ushape}(\Delta_2(\text{main})) = \text{sty}(\text{main})$. It is easy to see that check can be given the type $\Delta_2(\text{check})$. We show that app can be given the type $\Delta_2(\text{app})$. We type the body of app under the type environment $\Delta_2: a:\text{int}, f: \{u\mid u \geq a\} \rightarrow \ast, x: \{u\mid u \geq a\}$. Therefore, the application $f\ x$ type-checks to give the type $\ast$. To type the other branch, app $(x + 1)$, we instantiate $\Delta_2(\text{app})$ with $a$ at app to give the type $f:\{u\mid u \geq a\} \rightarrow \ast \rightarrow x: \{u\mid u \geq a\} \rightarrow \ast$.

The type is then used to give the application app $f$ the type $x: \{u\mid u \geq a\} \rightarrow \ast$, which in turn gives the branch the type $\ast$. Finally, main can be given the type $\Delta_2(\text{main})$ by instantiating $\Delta_2(\text{app})$ with $i$.

Likewise, let $\Delta_3$ be the following type environment.

$$
\begin{align*}
\Delta_3(\text{app}) &= \forall a: \text{int}. f: \tau_1 \rightarrow \forall b: \text{int}. g: \tau_2 \rightarrow \ast \\
\Delta_3(\text{app}) &= x: \text{int} \rightarrow \forall a: \text{int}. f: \{u\mid u \geq a\} \rightarrow \ast \rightarrow \ast \\
\Delta_3(\text{succ}) &= \forall b: \text{int}. f: \sigma \rightarrow \ast \rightarrow \ast \\
\Delta_3(\text{check}) &= x: \text{int} \rightarrow y: \{u\mid u \geq x\} \rightarrow \ast \\
\Delta_3(\text{main}) &= i: \text{int} \rightarrow \ast
\end{align*}
$$

such that $\tau_1 = \{u\mid u \geq a\} \rightarrow \ast$, $\tau_2 = \forall c: \text{int}. \tau_1 \rightarrow \ast$, and $\sigma = \{u\mid u \geq b\}$. Note that $\text{ushape}(F) = \text{sty}(\Delta_3(F))$ for each $F = \text{app3}$, $\text{app}$, $\text{succ}$, $\text{check}$, and $\text{main}$. We show that $\Delta_3 \vdash \Delta_3$ 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 $\text{succ}$ with $a$, and instantiating the resulting app3 ($\text{app}\ f$) with $a$ (or with an arbitrary side-effect-free integer-type expression, because $b$ does not appear free in its scope in $\Delta_3(\text{app3})$). 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 $\Delta_3(\text{app3})$). Then, to type main, we instantiate app3 with $i$, and instantiate app3 ($\text{check}\ i$) 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 adapted 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 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 $\text{Sub}$. 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 $\text{bvs}(\ell)$ 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 $\ell$ to a side-effect-free integer-type expression $\sigma$ such that $\tau(\ell) \subseteq \text{bvs}(\ell)$. We use $P(\ell)$ as the instantiation expression at $\ell$. 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

$$
\begin{align*}
c_0 + c_1 x_1 + \cdots + c_n x_n
\end{align*}
$$

where $\text{bvs}(\ell) = \{x_1, \ldots, x_n\}$ 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-

---

9 This can be seen as a difference between just checking safety and expressing the precise semantics as weakest preconditions.
The parameter substitution $P$ may be initialized arbitrarily, for example, $P(\ell) = 0$ for all $\ell$. We translate the input program by adding the extra parameters to the function definitions and replacing each instantiation site $e$ with the application $eP(\ell)$ where $\ell$ 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 $\ell_1$ and $\ell_2$ be the labels of the occurrences of $app$ in the body of $app$ and $main$ respectively. We have $L = \{\ell_1, \ell_2\}$, $bvs(\ell_1) = \{a, x\}$, and $bvs(\ell_2) = \{i\}$. Suppose that the parameter substitution is $P = \{\ell_1 \mapsto a, \ell_2 \mapsto i\}$. Then, $d_2$ is translated to the program $d_\rho$ shown below.

```plaintext
app a f x = if * then app a 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 i (check i)
i
```

**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). 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 $d_\rho$ from Example 4.1, MoCHi may generate the SHP $d_S$ shown below.

```plaintext
app1 a f x = app2 a f (x + 1)
app2 a f x = f x
check x y = assume ¬(x ≤ y); assert 0; check x y
main i = app i (check i)
i
```

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

$$\Delta(app1) = \Delta(app2) = a : \text{int} \rightarrow f : \{\{u \mid u ≥ a\} \rightarrow \star\} \rightarrow x : \{u \mid u ≥ a\} \rightarrow \star$$

$$\Delta(check) = x : \text{int} \rightarrow y : \{u \mid u ≥ x\} \rightarrow \star$$

$$\Delta(main) = i : \text{int} \rightarrow \star$$

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

**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 $\text{Sub}$ 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 Appendix F 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 $i$.

```plaintext
app1 a f x = app2 a f (x + 1)
app2 a f x = f x
check x y = assert ¬(x ≤ y); assert 0; check x y
main i = app i 0 (check i)
i
```

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 $P_R$ for the instantiation sites of $d'_S$, and...
from it, obtain the SHP template $d_F$ shown below.

\[
\begin{align*}
\text{app1: } & f \ x \Rightarrow \text{app2} (p_0 + p_1 a + p_2 x) f (x + 1) \\
\text{app2: } & f \ x \Rightarrow f \\
\text{check: } & x = y \Rightarrow \neg (x \leq y); \text{assert 0}
\end{align*}
\]

\[\begin{align*}
\text{main: } & \text{app1}(p_1 + p_4)(\text{check})\ i
\end{align*}\]

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

\[
\begin{align*}
\Delta(\text{app1}) &= a : \{u \mid P_1(u) \} \rightarrow \{u \mid P_2(a, u) \} \rightarrow \star \rightarrow \{u \mid P_3(a, u) \} \rightarrow \star \\
\Delta(\text{app2}) &= a : \{u \mid P_1(u) \} \rightarrow \{u \mid P_5(a, u) \} \rightarrow \star \rightarrow \{u \mid P_6(a, u) \} \rightarrow \star \\
\Delta(\text{check}) &= x : \{u \mid P_7(u) \} \rightarrow y : \{u \mid P_8(x, u) \} \rightarrow \star \\
\Delta(\text{main}) &= \text{int} \rightarrow \star
\end{align*}
\]

This generates the following non-recursive set of Horn-clause-like constraints on the predicate variables $P_1, \ldots, P_9$.

\[
\begin{align*}
\forall a, x. & P_1(a) \land P_2(a, x) \Rightarrow P_3(p_0 + p_1 a + p_2 x) \\
\forall a, x. & P_1(a) \land P_3(a, x) \land P_5(p_0 + p_1 a + p_2 x, u) \Rightarrow P_2(a, u) \\
\forall a, x, P_1(a) \land P_3(a, x) \Rightarrow P_4(p_0 + p_1 a + p_2 x, x + 1) \\
\forall a, x, P_1(a) \land P_6(a, x) \Rightarrow P_5(a, x) \\
\forall x, y, P_7(x) \land P_8(x, y) \Rightarrow y \leq x \\
\forall i, \top \Rightarrow P_9(p_3 + p_4 i) \\
\forall i, u, P_9(p_3 + p_4 i, u) \Rightarrow P_8(i, u) \\
\forall i, \top \Rightarrow P_9(p_3 + p_4 i, i)
\end{align*}
\]

Systematically simplifying the constraints by computing the least integer variables as bit-vectors and integer operations such that $\Delta$ is a quantifier-free non-linear FOL formula on the variables $p_0, \ldots, p_4$ shown below.

\[
\forall x, y, z. \\
p_4 x = p_4 z \land p_1 p_4 x + p_2 z = (p_1 p_4 + p_2)(y - 1) \Rightarrow x \leq y
\]

### 4.2.2 Constraint Solving

The constraint generation process above returns a constraint of the form $\forall x, y \in \mathbb{Z}, \theta$, where $\theta$ is a quantifier-free non-linear FOL formula over the variables $\overline{p}$ and $\overline{x}$, (More precisely, $\theta$ is linear over $\overline{x}$ with coefficients over $\overline{p}$.) The goal of the constraint solving phase is to find an assignment $\rho$ for $\overline{p}$ satisfying $\forall x, y \in \mathbb{Z}, \theta$. We solve the constraints by adopting Gullwani 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 $\forall x, y \in \mathbb{Z}, \theta$, and obtain a constraint of the form $\exists x, \overline{\rho}, \phi$, where $\phi$ is a quantifier-free non-linear FOL formula on the variables $\overline{p}$ and $\overline{q}$. 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 $\rho$ for $\overline{p}$ and $\overline{q}$ that satisfies $\phi$.

We describe the approach in more detail. Given a universally quantified constraint $\forall x, y \in \mathbb{Z}, \theta$, we convert $\theta$ to an equivalent universally quantified formula of the form $\bigwedge_i \theta_i$, where each $\theta_i$ is of the form $\neg \bigwedge_{j=1, \ldots, m_i} c_{i,j} \geq 0$ such that each $c_{i,j} \geq 0$ is a linear inequality on $\overline{x}$ with polynomials on $\overline{p}$ as coefficients. We translate each $\theta_i$ by applying Farkas’ lemma.

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

\[
\begin{align*}
& \begin{cases}
  c_{1,0} + c_{1,1} x_1 + \ldots + c_{1,n} x_n \geq 0 \\
  \vdots \\
  c_{m,0} + c_{m,1} x_1 + \ldots + c_{m,n} x_n \geq 0
\end{cases}
\end{align*}
\]

The system is unsatisfiable if there exist non-negative reals $r_0, r_1, \ldots, r_n$, such that

- $r_0 + r_1 c_{1,0} + \cdots + r_n c_{m,0} = -1$, and
- $r_j c_{1,j} + \cdots + r_n c_{m,j} = 0$ 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 $d_F$ in Example 4.3. We translate the constraints to the equivalent form below.

\[
\begin{align*}
\forall x, y, z. \\
- p_4 x & \leq 0 \\
- p_4 x & \geq 0
\end{align*}
\]

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

\[
\begin{align*}
\exists r_0, r_1, r_2, r_3, r_4 \geq 0. \\
r_0 + (p_1 p_4 + p_2)(r_3 - r_4) - r_5 = -1 & \land \\
r_1 (r_3 - r_2) + p_1 p_4 (r_4 - r_3) + r_5 = 0 & \land \\
-p_1 p_4 + p_2 (r_3 - r_4) = 0 & \land \\
-p_1 (r_1 - r_2) + p_2 (r_3 - r_4) = 0
\end{align*}
\]

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

\[
\begin{align*}
p_0 = 0, & \quad p_1 = 1, \quad p_2 = 0, \quad p_3 = 0, \quad p_4 = 1 \\
r_0 = 1, & \quad r_1 = 0, \quad r_2 = 0, \quad r_3 = 0, \quad r_4 = 1, \quad r_5 = 1
\end{align*}
\]

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 $\rho$ to $d_F$ results in the program equivalent to $d_S$ from Example 4.2. Finally, we update the parameter substitution $P$ via $P_F P$ to translate the input program, which results in the program $d_F$, 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 Gullwani 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 $P_F P$ is inferred, it is better to infer candidate types for the translated counterexample SHP $d_F P$ 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
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.\(^\text{12}\) (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 typable 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.\(^\text{13}\) (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 \(|\vec{p}|\) is the total number of the variables \(\vec{p}\) that occur in the range of the parameter substitution template \(P_{\vec{t}}\) (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.

| program | S | O | \(|\vec{p}|\) | \#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\% | |
| fnhnn | 26 | 2 | 1 | 1 | 1 | 0.15s | 12\% | | ✓ |
| repeat-add | 51 | 2 | 5 | 1 | 3 | 1.05s | 7\% | |
| app-leq | 29 | 2 | 1 | 1 | 1 | 0.26s | 41\% | ✓ |
| app-lin-ord2 | 37 | 2 | 1 | 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.

refinement type systems proposed for automated higher-order program verification [13, 18, 23, 26, 27]. We describe each program.

- \(d_2\) and \(d_3\) are from Example 3.4.
- fnhnn compares the return value of two function arguments:
  ```ocaml
  let f x y = assert (x () = y ()) in
  let h x () = x in
  let main n = f (h n) (h n)
  ```
  \(\text{repeat-add}\) makes the closure add \(n\) for some \(n \geq 0\), applies it to 0 for \(k\)-times for some \(k > 0\), and asserts that the result is not less than \(n\):
  ```ocaml
  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 \(e = (a = b)\) (resp. \(e = (4*a+2*b)\)).
  ```ocaml
  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 \(i\) for some non-deterministically-chosen \(i \geq n\):
  ```ocaml
  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:
  ```ocaml
  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:

```plaintext
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 \( a + b \):

```plaintext
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 `1-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:

```plaintext
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

- **1-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.
- **1-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 \( len1 + len2 \).
- **1-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 untyable 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

\[ \text{equals the sum of the first and the second elements of the function-encoded array passed as the first argument.} \]

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:

\[ \forall a \cdot \forall \theta \cdot b : \text{int}. \]

\[ (\text{int} \times (x : \text{int} \rightarrow \{ \theta \})) 
  \rightarrow \{ u \mid u = a + b \} \rightarrow \ast \]

where \( \theta \equiv (x = 0 \Rightarrow u = a) \land (x = 1 \Rightarrow u = b) \), and \( x \) 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 untyable 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 \( i \).

\[ \forall i : \text{int}. f : ((u | u = i) \rightarrow \ast) \rightarrow x : (u | u = i) \rightarrow \ast \]

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:

\[ \forall i : \text{int}. f : ((u | u = c) \rightarrow \ast) \rightarrow x : (u | u = c) \rightarrow \ast \]

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 \( \Pi \) 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, despite 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
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 \( \exists \vec{x}. \forall \vec{y}. \vartheta \) where \( \vartheta \) is a quantiﬁer-free arithmetic formula over the variables \( \vec{x} \) and \( \vec{y} \), and if valid, ﬁnding appropriate instantiations for \( \vec{x} \) so that the formula can be reduced to \( \forall \vec{y}. \vartheta[\vec{x}/\vec{x}] \).

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 program 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

A. Simple Type System

\[ \Gamma \vdash e : \kappa : \Gamma(\kappa) \quad \Gamma \vdash c : \text{cty}(c) \]
\[ \Gamma \vdash e_1 : s \quad \Gamma, x : s \vdash e_2 : \ast \]
\[ \Gamma \vdash x = e_1 \text{ in } e_2 : \ast \quad \Gamma \vdash e : s \rightarrow s' \quad \Gamma \vdash y : s' \]
\[ \Gamma \vdash x : \text{int} \quad \Gamma \vdash e_1 : \ast \quad \Gamma \vdash e_2 : \ast \]
\[ \Gamma \vdash \text{if } x \text{ then } e_1 \text{ else } e_2 : \ast \]

Figure 11. The simple type system typing rules.

The syntax of simple types is defined by the following grammar:
(Also shown in Section 3.)
\[ s ::= * \mid \text{int} \mid s \rightarrow s' \]

Figure 11 shows the typing rules of the simple type system. Here, \( \text{cty}(c) \) is the simple type of the constant \( c \). We overload the symbols \( \Gamma \) and \( \Delta \) to range over mappings to simple types.

DEFINITION A.1 (Simply-typed program). Let \( \Delta \) be a top-level simple type environment. We write \( \Gamma \vdash^* d \) if for each function \( F : \tau \rightarrow e \in d \), we have \( \Delta, \tau \vdash^* e : \ast \) where \( \Delta(F) = \tau \rightarrow \ast \).

We say that a program \( d \) is simply typed if there exists \( \Delta \) such that \( \Delta \vdash^* d \).

The simple type of a sub-expression \( e \) of \( d \), \( \text{sty}(e) \), is defined to be the type assigned to the (unique) occurrence of \( e \) in the derivation \( \Delta \vdash^* d \).

The following theorem is the standard type safety result that says that a simply-typed program does not get stuck, which can be proven via the standard method [29]. (Recall that due to CPS, a program cannot return normally.)

THEOREM A.2 (Simple Type Safety). Let \( d \) be simply-typed and main \( \tau = \text{main} \in d \). Then, for any arguments \( v \) of main and any \( e \) such that \( \text{main}(\tau \rightarrow \ast) \rightarrow \ast \), either \( e \) fail, \( e \) safe, or there exists \( e' \) such that \( e \rightarrow_\ast e' \).

B. Sound and Precise Constant Types

For a constant operation \( c \) over integers, we say that the type \( \tau = x:\text{int} \rightarrow \{u \mid \phi\} \) is sound if the following properties all hold:
- \( \tau \) is closed.
- \( \text{tshape}(\tau) = \text{cty}(c) \). (cf. Appendix A.)
- For any \( i \rightarrow j \) in the input domain of \( c \), \( \phi[i / \tau] \mid [c(i / \tau)]/u = \top \).

We say that \( \tau \) is precise if in addition to the above, \( \phi[i / \tau] \mid [j/u] = \bot \) for \( j \neq [c(i / \tau)] \). For example, for a 0-ary operators (i.e., constant integer) \( c \), the precise types are of the form \( \{u \mid u = c\} \).

While a non-assert constant operator in our language accepts all inputs in its domain, behaviors such as the divide-by-zero error can be modeled by inserting appropriate assert statements.

C. Typing \text{pureExps}

We add the rule below to type (non-A-normal-form) side-effect-free integer-type expressions.
\[ \Gamma \vdash e : \sigma \rightarrow \tau \quad \Gamma \vdash o : \sigma \]
\[ \Gamma \vdash e : \tau[o/y] \]

Alternatively, we may A-normalize the expressions and use the typing rules from Section 3.1.

D. Proof of Theorem 3.6

Theorem 3.6: If \( \Delta \vdash v \) then \( d \) is safe.

The proof is by the usual syntactic approach [29] and is similar to those of the previous work on refinement type systems [23, 27].
Therefore, it suffices to prove Theorem 3.6 for the stack semantics.

We extend the typing rules to type the stack augmented intermediate expressions. (We equate \( \text{st} \in e \) with \( e \).)

\[
\Gamma \vdash \gamma : \sigma \quad \Gamma, a : \sigma \vdash \ell \in e : \star \\
\Gamma \vdash \text{st} \ (a = v), \ell \in e : \star \\
\text{St}
\]

We also add the following so that \( \text{safe} \) is always typable.

\[
\Gamma \vdash \gamma \text{ safe} : \star \\
\text{Safe}
\]

The following is immediate from the property of sound constant types (Appendix B).

**Lemma D.2.** Suppose \( a \in \text{pureExp} \) and \( \Gamma \vdash a : \{ u \mid \theta \} \). Then, \( \llbracket \Gamma \rrbracket \rightarrow \exists u. \theta \).

The following lemma states that to we can derive \( \Gamma \vdash \gamma \in e : \star \) by binding a fresh variable bound to the type of a side-effect-free integer-type expression in scope.

**Lemma D.3.** Suppose \( \Gamma \vdash \gamma : \tau \) and \( \Gamma, x : \tau \vdash \gamma : \star \) such that \( a \in \text{pureExp} \) and \( x \not\in \text{fv}(e) \). Then, \( \Gamma \vdash \gamma : \star \). (Note that we assume \( x \notin \text{dom}(\Gamma) \).)

Proof: By inspection of the \( \gamma \) typing rules, the binding of the non-occurring variable \( x \) can affect the type derivation of \( e \) in only two ways: (1) by using \( x \) as a sub-expression of the instantiating expression at \( \text{Inst} \), or (2) by having \( x : \tau \) in the left hand side of \( \Rightarrow \) within the FOL formula denoting the assumptions about the base-type variables in the environment at the subtyping judgement \( \text{Sh} \).

In the case (1), any such use of \( x \) can be substituted by \( y \) to attain the same effect. In the case (2), suppose \( \tau = \{ u \mid \theta \} \). Then, in \( \text{Sh} \), \( \theta \) may only occur as \( \phi_i \land \theta\{x/u\} \Rightarrow \phi_2 \) such that \( x \) does not occur free in \( \phi_1, \phi_2, \) or \( \theta \). Therefore, the formula is logically equivalent to \( \phi_i \land \exists u. \theta \Rightarrow \phi_2 \). By inspection of the typing rules, \( \phi_i \) must be such that \( \phi_i \Rightarrow \llbracket \Gamma \rrbracket \). Also because \( \Gamma \vdash \gamma : \{ u \mid \theta \} \) and \( \alpha \in \text{pureExp} \), it follows that \( \llbracket \Gamma \rrbracket \rightarrow \exists u. \theta \) by Lemma D.2. And, we can derive the instance of \( \text{Sh} \) without the binding. \( \Box \)

We state a couple of substitution lemmas.

**Lemma D.4.** (Substitution I). Suppose \( \Gamma \vdash \gamma : \sigma \) and \( \Gamma, x : \sigma \vdash \gamma \in e : \star \). Then, \( \Gamma \vdash \gamma[e/x] : \star \).

Proof: By induction on the derivation. \( \Box \)

**Lemma D.5.** (Substitution II). Suppose \( \Gamma \vdash \gamma : \sigma_1 \), and \( \Gamma, x : \sigma_1 \vdash \gamma \in e : \star \). Then, \( \Gamma \vdash \gamma[\sigma_1/e] \leq \sigma_3 \).

Suppose \( \Gamma \vdash a : \sigma_1 \) and \( \gamma \in e : \star \). Then, \( \Gamma \vdash \gamma[a/x] \leq \sigma_3 \).

Proof: By induction on the derivation. \( \Box \)

Given type environments \( \Gamma_1 \) and \( \Gamma_2 \), let \( \Gamma_1, \Gamma_2 \) denote the type environment formed by appending the two in sequence in that order. It suffices show the following preservation lemma to prove the theorem.

**Lemma D.6.** (Preservation). Suppose \( \Delta \vdash \gamma \in e : \star \) and \( \ell \vdash \gamma \rightarrow d \). Then, \( \Delta \vdash \gamma : \star \).

Proof: We prove by case analysis on the reduction rule.

Suppose \( e \) is of the form \( \text{st} \in e \) in \( \Gamma \) and \( \Gamma \) is neither an assume nor assertion expression. Then, \( \Delta \vdash \gamma : \star \) and \( \ell \vdash \gamma \rightarrow d \) imply that we must have \( \Delta, \Gamma, x : \sigma \vdash e : \star \) and \( \ell(e_1) \) and \( \Gamma \) is the bindings from the stack \( \ell \) formed by \( \text{St} \). Therefore, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \text{st} \ (b = v), \ell \in e_2 \). Hence, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e_2 : \star \).

Suppose \( e \) is of the form \( \text{st} \in e \) in \( \Gamma \) and \( \Gamma \) is neither an assume nor assertion expression. Then, \( \Delta \vdash \gamma : \star \) and \( \ell \vdash \gamma \rightarrow d \) imply that we must have \( \Delta, \Gamma, x : \sigma \vdash e : \star \) and \( \ell(e_1) \) and \( \Gamma \) is the bindings from the stack \( \ell \) formed by \( \text{St} \). Therefore, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e_1 : \star \).

Therefore, it must be the case that \( (a = 1) \in \ell \), and we have \( e \rightarrow_d \ell \). Hence, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e : \star \).

By induction on the derivation.

For each \( i \), let \( \gamma_i \) be the \( \gamma \) of \( \text{inst} \) at \( \ell \) and \( \gamma_i \) is the \( \gamma \) of \( \text{Inst} \) at \( \ell \).

Therefore, if \( (a = 1) \in \ell \), then \( e \rightarrow_d \ell \).

Suppose \( e \) is of the form \( \text{st} \in e \) in \( \Gamma \) and \( \Gamma \) is neither an assume nor assertion expression. Then, \( \Delta \vdash \gamma : \star \) and \( \ell \vdash \gamma \rightarrow d \) imply that we must have \( \Delta, \Gamma, x : \sigma \vdash e : \star \) and \( \ell(e_1) \) and \( \Gamma \) is the bindings from the stack \( \ell \) formed by \( \text{St} \). Therefore, if \( (a = 1) \in \ell \), then \( e \rightarrow_d \ell \).

Therefore, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e_1 : \star \).

By induction on the derivation.

Therefore, if \( (a = 1) \in \ell \), then \( e \rightarrow_d \ell \). Hence, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e_1 : \star \).

By induction on the derivation.

Therefore, if \( (a = 1) \in \ell \), then \( e \rightarrow_d \ell \). Hence, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e_1 : \star \).

By induction on the derivation.

Therefore, if \( (a = 1) \in \ell \), then \( e \rightarrow_d \ell \). Hence, by Lemma D.4 and \( \text{St} \), we have \( \Delta \vdash \gamma \in e_1 : \star \).

By induction on the derivation.
where $c \rightarrow t$ is a total application of a non-assert/assert constant. Note that any value is a term. Given a term $t$ with free variables $\overline{x}$, we define the evaluation relation $(t)_{\overline{x}} v$ of $v$ in the obvious way: $v \in (t)_{\overline{x}} v$ with all constant operator applications in the subexpression reduced to integer constants.

For a value $v$, we write $\lceil v \rceil$ to denote the Gödel number that encodes $v$. We let $\lceil i \rceil = i$ for an integer constant $i$. (Collisions between function-type and integer-type values can always be avoided in our setting because the program is assumed to be simply typed.) For each variable $x$, we introduce a distinct base-type (i.e., integer-type) variable $\overline{x}$. We write $\overline{x}$ for the sequence $x_1, \ldots, x_n$ where $\overline{x} = x_1, \ldots, x_n$. We write $\alpha(t)$ for the set of variables (\overline{x}) where $\alpha(t) = \{t\}$.

For a term $t$, we define $\lceil t \rceil$ to be the integer-type term of $t$ that encodes $t$. More formally, $\lceil t \rceil$ is the integer term that satisfies

$$\forall v, \overline{x}. (t)_{\overline{x}} v \Leftrightarrow \lceil t \rceil (\overline{x}) = \lceil t \rceil$$

where $\overline{x} = \alpha(t)$, and $\lceil \overline{x} \rceil$ is the sequence $\lceil v_1 \rceil, \ldots, \lceil v_n \rceil$ where $\overline{x} = v_1, \ldots, v_n$.

Let $\text{maxargs}$ be the maximum number of function arguments of $d$, that is, $\text{maxargs} = \max \{ | \overline{x} | : F \overline{x} = d \}$. Without loss of generality, we assume that each function of $d$ has the same function argument names, that is, there exists $\overline{d}$ such that $| \overline{d} | = \text{maxargs}$ and for each $F \overline{d} = d$, $\overline{d}$ is a prefix of $\overline{d}$. (Note that the variables of the same name in different functions need not have the same simple type.) In the rest of the proof, we fix the variables $\overline{d}$ to be the sequence of function arguments of the program $d$. Let $\overline{d} = \overline{d}$. We also introduce a special base-type variable $x_{\overline{d}}$ that is distinct from $\overline{d}$.

Let $\psi$ be a formula such that $\psi(t) \subseteq \{ x_{\overline{d}}, \overline{d} \}$. We intend to use $\psi(x_{\overline{d}}, \overline{d})$ to express the condition for which it is safe to call the function encoded by $x_{\overline{d}}$ via the arguments encoded by $\overline{d}$. We say that an expression is a CPS expression if it is given the simple type $* \ (e.g., a function body). Figure 13 shows the rules for deriving $\text{wpre}(e, \psi)$ for a CPS expression $e$. Intuitively, $\text{wpre}(e, \psi)$ is the condition safe to evaluate $e$ assuming that $\psi$ is the condition safe to call the functions. Note that $\text{wpre}(e, \psi) \subseteq \{ \overline{d} \}$.

Let $\overline{x}_{\text{in}}$ denote the first $m$ elements of the sequence $\overline{x}$, and $\overline{d}_{\text{in}}$ denote the sequence of elements from the $m$th element to the $n$th element of $\overline{d}$. We define the weakest precondition of the program by taking the fixed point of the following predicate.

$$\text{step}(\psi) \equiv \forall F, \overline{d}. \exists F \overline{d} = x_{\overline{d}} \Rightarrow \text{wpre}(eF, \psi)(\overline{d} / \overline{d})$$

where $F, \overline{d} = eF \in d$, $\overline{d} = \overline{d}_{\text{maxargs}}$, and $\overline{d} = \overline{d}_{\text{maxargs} \overline{d}}$. Note that $\text{wpre}(\text{step}(\psi)) \subseteq \{ x_{\overline{d}}, \overline{d} \}$.

We write $e \rightarrow c$ if $c$ can be reduced to $e'$ by making at most $n$ function calls. We show the correctness of the weakest precondition predicate transformer.

**Lemma E.1.** Suppose $\psi = \text{step}^n(\psi)$.

1. Let $e$ be a CPS sub-expression of $d$, $\psi(e) = \{ \overline{d} \}$, and $\overline{d}$ be values such that $e(\overline{d} / \overline{d})$ is simply typed. Then, $e(\overline{d} / \overline{d}) \not\equiv \psi(e, \psi)$ fail.

2. Let $F \overline{d}, \overline{d} \equiv \text{a simply-typed total application. Then,}$

$$F \overline{d}, \overline{d} \not\equiv \psi(F \overline{d}, \overline{d})$$

**Proof:** We prove by induction on $n$ and $e$. First, we show (1) by induction on $e$, assuming that (2) holds.

Suppose $e$ is let $y = \text{in} e_1$. Then, $e(\overline{d} / \overline{d}) \equiv \psi(e, \psi)$ fail if $e_1(v/y) \not\equiv \psi(e, \psi)$ fail if $\theta[\overline{d} / \overline{d})$ is valid where $\theta = \text{wpre}(e_1, \psi)$. Therefore, $e(\overline{d} / \overline{d}) \not\equiv \psi(e, \psi)$ fail if $\theta[\overline{d} / \overline{d})$ is valid.

Suppose $e$ is let $y = \text{assume} e_2$. Then, $e(\overline{d} / \overline{d}) \equiv \psi(e, \psi)$ fail if $e_2(\overline{d}/\overline{d}) \not\equiv \psi(e, \psi)$ fail if $\theta[\overline{d} / \overline{d})$ is valid where $\theta = \text{wpre}(e_2, \psi)$. Therefore, $e(\overline{d} / \overline{d}) \not\equiv \psi(e, \psi)$ fail if $\theta[\overline{d} / \overline{d})$ is valid.

Suppose $e$ is a simply-typed application $t. Let e(\overline{d} / \overline{d}) = F \overline{d}$. Then, by (2), $F \overline{d} / \overline{d} \not\equiv \psi(F \overline{d}, \overline{d})$ (i.e., $\theta[\overline{d} / \overline{d})$ is valid.

Next, we show (2) by induction on $n$. The case $n = 0$ is trivial. Suppose that (1) and (2) hold for $n = 1$. Let $F \overline{d} = eF \in d$. Then, by (1), $F \overline{d}, \overline{d} \not\equiv \psi(F \overline{d}, \overline{d})$ is valid where $\theta = \text{wpre}(eF, \psi)$ and $\psi' = \text{step}^{n-1}(\psi)$, Therefore, $\text{step}(eF, \psi) \not\equiv \psi(F \overline{d}, \overline{d})$ (i.e., $\theta[\overline{d} / \overline{d})$ is valid.

The least fixed point of $\text{step}$ is the following formula. (Least in the order of logical implication.)

$$\Psi \equiv \forall \exists \text{step}^n(\psi)$$

Note that $\Psi$ can be expressed as a FOL formula by using Gödel numbering to encode the sequence of step applications. (This requires encoding the FOL formulas in the sequence.)

The following is immediate from the definition of $\Psi$.

**Lemma E.2.** Let $t_1, t_2$ be a simply-typed application, and $\overline{d} = x_{\overline{d}}, \overline{d}$. Then,

$$\Psi(\overline{d}_1, \overline{d}_2) / x_{\overline{d}}[\overline{d} / \overline{d}]) \equiv \Psi(\overline{d}_1, \overline{d}_2)[\overline{d}_2 / x_{\overline{d}}]$$

Note that $\Psi$ can be expressed as a FOL formula by using Gödel numbering to encode the sequence of step applications. (This requires encoding the FOL formulas in the sequence.)
We prove a few auxiliary lemmas.

**Lemma E.8.** Let $t_1 t_2$ be a simply-typed partial application. Let $\tau_1 = \text{wp}(t_1)$.

1. If $\text{sty}(t_2) = \text{int}$, then $\tau_2 = x : \text{int} \to \sigma$ such that $\sigma[[t_2]/x] = \text{wp}(t_1 t_2)$.
2. Otherwise, $\text{sty}(t_2) \in \text{args}$, and $\text{wp}(t_1) = \forall x : \text{int} . y : \tau \to \sigma$ such that $\tau[[t_2]/x] = \text{wp}(t_2)$ and $\sigma[[t_2]/x] = \text{wp}(t_1 t_2)$.

**Proof:** We show (1). By the construction of $\text{wp}(t)$ and the fact that $t_1 t_2$ is a partial application, it follows that $\tau_1 = \text{wp}(t_1)$. Suppose the last argument of $\tau$ and $\sigma[[t_2]/x_1]$ is a base type. From the construction, it is easy to see that $\tau$ and $\sigma[[t_2]/x]$ are alpha-equivalent, except for the refinement predicate of a non-last base-type argument to $\tau$. Let $\sigma$ have $n$ arguments so that $\sigma = x : \tau_1 \rightsquigarrow \ast$ with $i = 2, \ldots, n + 1$, and $x_1 :: \tau_1$ is a prefix of $\tau$. Let $\tau = x_1, \ldots, x_n$ also be the arguments of $\tau$ (i.e., via alpha-renaming). Let $\theta_t$ and $\theta_\sigma[[t_2]/x_1]$ be the refinement predicates in the last argument of $\tau$ and $\sigma[[t_2]/x_1]$, respectively (i.e., $\theta_t$ is the refinement predicate in the last argument of $\sigma$). Then, we have

\[ \theta_t = (\forall x_1. \text{wp}_t[[t_1 t_2]/x_1])_{[u/x]} \]

\[ \theta_\sigma[[t_2]/x_1] = (\forall x_1. \text{wp}_\sigma)[[t_1 t_2]/x_1][u/x_0]][u_{x_0 + 1}/x_1] \]

where $Y_\sigma = \text{arg}^\ast (\text{sty}(t_1 t_2))$, and $Y_\sigma = \text{arg}^\ast (\text{sty}(t_1 t_2))$. Therefore, by Lemma E.2, it follows that

\[ \theta_t[[t_2]/x] \Leftrightarrow \theta_\sigma[[t_2]/x] \]

Therefore, $\sigma[[t_2]/x] = \text{wp}(t_1 t_2)$ (via alpha-renaming).

Otherwise, the last argument of $\tau$ and $\sigma[[t_2]/x_1]$ is a function type (of the same simple type), and an analogous argument holds for the refinement predicate in the type of the universally bound variable just before the argument.

We show (2). By the construction of $\text{wp}(t)$ and the fact that $t_1 t_2$ is a partial application, it follows that $\tau_1 = \text{wp}(t_1)$ is of the form

\[ \forall x : \text{int} . y : \tau \to \sigma \] (i.e., because the $\text{wp}(t)$ construction sets the refinement predicate of a universally bound variable just before a non-last function-type argument to $\tau$). From the construction, it follows that $\tau[[t_2]/x] = \text{wp}(t_2)$. And, $\sigma[[t_2]/x] = \text{wp}(t_1 t_2)$ follows from the argument similar to (1). }

**Lemma E.9.** Suppose $\Gamma \vdash_\forall e : \tau$ and $[[\Gamma]] \Rightarrow x = t$ Then, $\Gamma \vdash_\forall e : \tau[t/x]$. 


Proof: By Sub and the fact that if \( \theta \Rightarrow \psi \) and \( \theta \Rightarrow x = t \) then \( \theta \Rightarrow \psi[t/x] \). □

Note that for any term \( t\in \text{pureExps} \), we are now ready to state the main lemma.

**Lemma E.10.** Let \( e \) be a CPS-sub-expression. Let \( \rho \) be the substitution such that \( \rho(x) = x \) if \( x \) is a base-type variable in \( e \) and \( \rho(x) = \bar{x} \) if \( x \) is a function-type variable in \( e \). Let \( \Gamma \) be a type environment such that \( \Gamma \Gamma \Rightarrow \wpty(e, \Psi_{wp}) \rho \), and for each function-type variable or function name \( \kappa \in \text{dom}(\Gamma) \), there exists a term \( s \) such that \( \forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \). Therefore, by Lemma E.2,

(A) For each integer-type sub-term \( t \) of \( e \), \( \Gamma \vdash t : \{ u \mid \{ t \} \rho = u \} \).

(B) For each function-type sub-term \( t \) of \( e \), \( \Gamma \vdash t : \wpty(t) \).

(C) \( \Gamma \vdash e : \ast \).

**Proof:** (A) follows from the definition of precise constant types (cf.

(Appendix B).

We show (B) by induction on \( t \). The base case (i.e., \( t = x \) or \( t = F \)) is immediate from the assumption. So, let \( t = t_1 t_2 \). By induction hypothesis, we have \( \Gamma \vdash t_1 : \wpty(t_1) \). If \( t_2 \) is an integer-type term, then the result follows from (A), Lemma E.8 (1), and Lemma E.9. Otherwise, \( t_2 \) is a function-type term, and the result follows from Lemma E.8 (2), applying \( \text{Inst} \) with the instantiating expression \( \{ t_2 \} \) to instantiate \( t_1 \)'s type to \( \wpty(t_2) \rightarrow \wpty(t_1 t_2) \).

We show (C) by induction on \( e \). Suppose \( e \) is \( \text{let } x = t \text{ in } e_1 \). If \( e_1 \) is a base type, then by (A), we have

\[
\forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(e_1, \Psi_{wp}) \rho
\]

where \( \Gamma \vdash t : \tau \). Therefore, \( \forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(e_1, \Psi_{wp}) \rho \).

Also, \( \forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(e_1, \Psi_{wp}) \rho \Rightarrow y = 1 \). Therefore, by (B) and the type of \( \text{assert} \) (i.e., \( \{ u \mid u = 1 \} \rightarrow \{ u \mid u = 1 \} \)), we have \( \Gamma \vdash e : \ast \). The case is \( \text{let } x = \text{assume } y \text{ in } e_1 \) is similar.

Suppose \( e \) is \( \text{if } \cdot \text{ then } e \text{ else } e_2 \). We have

\[
\forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(e, \Psi_{wp}) \rho
\]

Therefore, by induction hypothesis, we have \( \Gamma \vdash e_1 : \ast \) and \( \Gamma \vdash e_2 : \ast \). Then, by (B), \( \Gamma \vdash e : \ast \).

Suppose \( e \) is a total application \( t \rightarrow t' \). Let \( t \rightarrow t_1, \ldots, t_n, \) and \( \theta = t \). By (B), we have \( \Gamma \vdash t' : \wpty(t') \). Suppose \( \text{sty}(t_n) = \int \). Then, by (A), we have that \( \Gamma \vdash t_n : \{ u \mid \{ t_n \} \rho = u \} \).

By the construction of \( \wpty(t') \), we have

\[
\forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(e, \Psi_{wp}) \rho \Rightarrow \text{arg} \Rightarrow \text{sty}(t') = \{ x \} \setminus \{ x \}
\]

where \( \text{sty}(t') = \{ x \mid \theta \} \rightarrow \ast \). Therefore, \( \Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \{ x \} \setminus \{ x \} \rho = u \Rightarrow \theta \). Hence, it follows that \( \Gamma \vdash e : \ast \).

The case \( \text{let } t \text{ in } e_1 \) is similar.

□

Also, the following is clear from the construction of \( \wpty(F) \).

**Lemma E.11.** Let \( F \) be \( \text{let } e \text{ in } d \). Then, \( \Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(F) \Rightarrow \ast \).

Then, \( \forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(F, \Psi_{wp}) \rho \), and

\( \forall \theta.(\Psi_{wp}(\Gamma, \kappa), \Gamma, s) \Rightarrow \wpty(F) \Rightarrow \ast \).

**F. Constraint Generation Process**

The constraint generation rules.

Figure 15 shows the constraint generation rules. Here, \( \text{cgen}(\Gamma \vdash e) \) returns a pair of a refinement type \( \tau \) and a constraint \( \theta \) such that \( \Gamma \vdash e : \tau \) is derivable if and only if \( \theta \) is valid. Similarly, \( \text{cgen}(\Gamma \vdash \tau_1 \leq \tau_2) \) returns a constraint \( \theta \) such that \( \Gamma \vdash \tau_1 \leq \tau_2 \) is derivable if and only if \( \theta \) is valid.