Dependent Types from Counterexamples [*]

Tachio Terauchi
Tohoku University
terauchi@ecei.tohoku.ac.jp

Abstract

Motivated by recent research in abstract model checking, we present a new approach to inferring dependent types. Unlike many of the existing approaches, our approach does not rely on programmers to supply the candidate (or the correct) types for the recursive functions and instead does counterexample-guided refinement to automatically generate the set of candidate dependent types. The main idea is to extend the classical fixed-point type inference routine to return a counterexample if the program is found untypable with the current set of candidate types. Then, an interpolating theorem prover is used to validate the counterexample as a real type error or generate additional candidate dependent types to refute the spurious counterexample. The process is repeated until either a real type error is found or sufficient candidates are generated to prove the program typable. Our system makes non-trivial use of "linear" intersection types in the refinement phase.

The paper presents the type inference system and reports on the experience with a prototype implementation that infers dependent types for a subset of the Ocaml language. The implementation infers dependent types containing predicates from the quantifier-free theory of linear arithmetic and equality with uninterpreted function symbols.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Type structure

General Terms Algorithms, Languages, Theory, Verification

Keywords Dependent types, Intersection types, Interpolation, Counterexamples, Type inference

1. Introduction

This paper follows the work on applying dependent types to checking complex properties of programs that are beyond the reach of conventional type systems like ML types. In this paper, by *dependent types*, we mean refinement types [14] that embed first-order logic formulas. For instance, suppose we want to check that the assertion never fails in the program shown in Figure 1. (Here, we use the Ocaml syntax.)

```
let rec mult x y =
   if x <= 0 || y <= 0 then
      0
   else
      x + mult x (y - 1)
in assert (100 <= mult 100 100)
```

Figure 1. The multiplication example.

One way to check the assertion is by giving `mult` the following dependent type.[1]

$$x : \mathtt{int} \rightarrow y : \mathtt{int} \rightarrow \{u : \mathtt{int} \mid (x \geq 0 \land y > 0 \Rightarrow u \geq x) \land (y \leq 0 \Rightarrow u \geq 0)\}$$

The type says that `mult` takes integers x and y, and returns an integer that is at least as large as x if $x \geq 0$ and $y > 0$, and non-negative if $y \leq 0$. (As usual, $\Rightarrow$ binds weaker than other logical operators, and $\rightarrow$ associates to the right.) Indeed, the type is a valid type for `mult` and is sufficient to prove that the assertion does not fail. Note that the type is neither the strongest (i.e., the most precise) type nor the weakest necessary type that can be assigned to `mult` to prove the assertion. The strongest type for `mult` would be

$$x : \mathtt{int} \rightarrow y : \mathtt{int} \rightarrow \{u : \mathtt{int} \mid (x > 0 \land y > 0 \Rightarrow u = x \times y) \land (x \leq 0 \lor y \leq 0 \Rightarrow u = 0)\}$$

which contains non-linear arithmetic, as expected.

This paper presents a method for inferring sufficiently strong dependent types to check the given properties of a program. Our approach avoids computing the strongest or the weakest necessary type, and instead returns some type that is sufficient to prove the property when terminating with success.

Many existing dependent type systems (e.g., [4, 10, 40]) require the programmer to annotate recursive functions like `mult` with the correct types. Other systems [14, 34] require the domain of candidate types to be pre-defined and form a finite-height lattice so that the type checking can be implemented as a fixed-point algorithm that infers the strongest types in a bottom-up manner.

We propose a different approach to checking and inferring dependent types that does not require a pre-defined set of correct or candidate dependent types. Our approach is inspired by research in counterexample guided abstraction refinement (CEGAR) for model checking [3, 9, 16, 28]. The core of our system is a CEGAR loop that iteratively refines the lattice of candidate dependent types until either the program is found to be actually untypable or the lattice

[*] This work was supported by MEXT KAKENHI 20700019 and 20240001.

[1] The type syntax is borrowed from Augustsson [1] (also used in [12, 22, 23, 34, 37]).

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01. . . $10.00

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

Figure 2. The syntax of the simple functional language.

becomes refined enough to type check the program. We start with a coarse lattice containing few candidates and gradually add types that are sufficient to refute the spurious counterexamples encountered during the CEGAR iteration.

A counterexample in our system is an "unwound" slice of the program that is untypable with the current candidates. The refinement phase decides whether the slice can be typed if the types are not confined to the candidates, and if so, generates new candidates from the inferred typing (if not, then the program is really untypable). For this, we employ recent techniques from both type systems and model checking research: *linear intersection types* [21] and *interpolation* [27]. We use linear intersection type inference to infer a type derivation "shape" that is sufficient for typing the counterexample, and we use an interpolating theorem prover to quickly compute good candidate types from the type derivation (in particular, without explicit quantifier elimination).

The rest of the paper is organized as follows. Section 2 introduces the language and the dependent type system. The main contribution of the paper is the CEGAR-inspired type inference system described in Sections 3, 4, 5, and 6. While the paper mostly focuses on the assertion checking application for simplicity, it is easy to extend the system to more general program specification checking as discussed in Section 7. Section 8 describes the prototype implementation for a subset of the Ocaml language, Section 9 discusses related work, and Section 10 concludes. The proofs of the key results appear in Appendix C.

2. Preliminaries

We focus on a small functional language shown in Figure 2. We briefly describe the syntax. A *program*, d, is a finite set of function definitions, $F\,\overrightarrow{x} = e$, which defines a function named F with the formal parameters $\overrightarrow{x}$ and the body e. The notation $\overrightarrow{a}$ denotes a possibly empty sequence. We often use letters u, x, y, z, u_i, etc. to range over program variables and first-order logic variables, and letters F, G, H, F_i, etc. to range over function names. Functions can be mutually recursive in that the body of a function may refer to other functions, including itself. We assume that each function is closed (except for the free function names). We also assume that every function name is unique and that there is a function named `main` that takes no arguments. Note that nested function definitions can be supported via lambda lifting [19].

An expression, e, is a variable, a constant c, a function name, a let expression $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$, a (constant or function) application $e\, x$, a conditional branch $\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, or an assertion $\texttt{assert}\ e$. Constants include integer and boolean constants such as 0 and $true$, as well as integer and boolean operations such as $+$ and $\leq$. For simplicity, we restrict branch condition and function arguments to just variables.[2]

We restrict the body of a function to continuation passing style (CPS) so that a function does not return. We also impose the CPS restriction to the body of a let expression (i.e., e' in $\texttt{let}\, x = e\, \texttt{in}\, e'$) and those of a branch (i.e., e_1 and e_2 in $\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$) so that they are also non-returning. As usual, non-returning expressions (i.e., non-partial function applications, let expressions, and conditional expressions) are restricted to occur only in a continuation context (i.e., not in a non-CPS evaluation context E shown in Figure 3). CPS is only enforced on user-defined functions so that constant operations need not be CPS.

$$E ::= \texttt{let}\, x = E\, \texttt{in}\, e \mid E\, e \mid v\, E$$

Figure 3. The non-CPS evaluation contexts.

$$\frac{F\,\overrightarrow{x} = e \in d \quad |\overrightarrow{x}| = |\overrightarrow{v}|}{F\,\overrightarrow{v} \to_d e[\overrightarrow{v}/\overrightarrow{x}]}\ \textbf{APP}$$

$$\frac{arity(c) = |\overrightarrow{v}|}{E[c\,\overrightarrow{v}] \to_d E[[\![c]\!](\overrightarrow{v})]}\ \textbf{CST}$$

$$\texttt{let}\, x = v\, \texttt{in}\, e \to_d e[v/x] \quad \textbf{LET}$$

$$\texttt{if}\ true\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to_d e_1 \quad \textbf{IF1}$$

$$\texttt{if}\ false\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to_d e_2 \quad \textbf{IF2}$$

$$E[\texttt{assert}\ true] \to_d E[0] \quad \textbf{AS1}$$

$$E[\texttt{assert}\ false] \to_d fail \quad \textbf{AS2}$$

Figure 4. The semantics of the simple functional language.

The CPS restriction is imposed only to simplify the exposition. Non-CPS expressions may be supported indirectly via CPS conversion or directly by extending the type system with conditional types and union (i.e., disjunctive) types.[3]

The rest of the syntax is straightforward. As usual, a function application associates to the left so that $e_0\, e_1\, e_2 = (e_0\, e_1)\, e_2$. We write $e_0\,\overrightarrow{e}$ for the series of applications $e_0\, e_1\, e_2\, \ldots\, e_n$ where $\overrightarrow{e} = e_1, e_2, \ldots, e_n$. We write $e_1; e_2$ for $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ such that $x \notin free(e_2)$. Without loss of generality, we assume that bound variables are distinct.

Note that, because of higher-order functions and partial applications, function calls are not syntactically obvious, and one syntactic occurrence of a function name may end up being called from multiple places.

We define the call-by-value semantics of the language as a small-step reduction relation from states to states. A *state* is a run-time expression e that extends the source expressions with values v and a special failure state, defined as follows.

$$
\begin{array}{l}
v ::= c \mid F \mid v_1\, v_2 \\
e ::= \cdots \mid v \mid fail
\end{array}
$$

(We overload the symbol e to range over run-time expressions when it is clear from the context.) We restrict the application $F\,\overrightarrow{v}$ (resp. $c\,\overrightarrow{v}$) to be a value only when it is partial, that is, only when the arity of F (resp. c) is greater than $|\overrightarrow{v}|$. Note that a partial application denotes a closure value.

Figure 3 defines the non-CPS evaluation contexts. Figure 4 shows the reduction rules. The reduction rules are mostly straightforward. In **CST**, the notation $arity(c)$ denotes the arity of the constant operation c. Here, $[\![c]\!]$ is the relation denoting the semantics of c, so that, for example $[\![+]\!](i, j) = i + j$ for all integers i and j. **AS1** returns a dummy value 0, and **AS2** aborts the program with an assertion failure. Note that, because of CPS, **APP**, **LET**, **IF1**, and **IF2** only occur at the top-level.

[2] The implementation lifts this restriction by online A-normalization [13].

[3] The latter approach is taken in the implementation discussed in Section 8.

$$\begin{array}{lcl}\tau & ::= & \star \mid \{u{:}B \mid \theta\} \mid x{:}\sigma \to \tau \\ \sigma & ::= & \tau \mid \sigma_1 \wedge \sigma_2\end{array}$$

Figure 5. The syntax of dependent types.

$$\frac{\mathit{sty}(x) \text{ is base}}{\Gamma;\theta \vdash x : \{u{:}B \mid u = x\}}\ \textbf{VaB} \qquad \frac{\mathit{sty}(x) \text{ is not base}}{\Gamma, x{:}\bigwedge_i \tau_i;\theta \vdash x : \tau_i}\ \textbf{VaF}$$

$$\frac{}{\Gamma, F{:}\bigwedge_i \tau_i;\theta \vdash F : \tau_i}\ \textbf{Fun} \qquad \frac{}{\Gamma;\theta \vdash c : \mathit{ty}(c)}\ \textbf{Cst}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma;\theta \vdash_\wedge e : \tau}\ \textbf{Int1} \qquad \frac{\Gamma_1;\theta \vdash_\wedge e : \sigma_1 \quad \Gamma_2;\theta \vdash_\wedge e : \sigma_2}{\Gamma_1 \wedge \Gamma_2;\theta \vdash_\wedge e : \sigma_1 \wedge \sigma_2}\ \textbf{Int2}$$

$$\frac{\Gamma_1;\theta \vdash_\wedge e_1 : \sigma \quad \Gamma_2, x{:}\sigma;\theta \vdash e_2 : \star}{\Gamma_1 \wedge \Gamma_2;\theta \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \star}\ \textbf{Let}$$

$$\frac{\Gamma_1;\theta \vdash e : y{:}\sigma \to \tau \quad \Gamma_2;\theta \vdash_\wedge x : \sigma' \quad \Gamma_2;\theta \vdash \sigma' \leq \sigma}{\Gamma_1 \wedge \Gamma_2;\theta \vdash e\, x : \tau[x/y]}\ \textbf{App}$$

$$\frac{\Gamma_1;\theta \wedge x = \mathit{true} \vdash e_1 : \star \quad \Gamma_2;\theta \wedge x = \mathit{false} \vdash e_2 : \star}{\Gamma_1 \wedge \Gamma_2;\theta \vdash \texttt{if}\, x\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \star}\ \textbf{If}$$

$$\frac{\Gamma;\theta \vdash_\wedge e : \sigma \quad \Gamma;\theta \vdash \sigma \leq \{u{:}\texttt{bool} \mid u = \mathit{true}\}}{\Gamma;\theta \vdash \texttt{assert}\, e : \{u{:}\texttt{int} \mid u = 0\}}\ \textbf{Assert}$$

Figure 6. The type checking rules.

We define a *run* of a program to be a sequence of reductions from the initial state $e_{\texttt{main}}$ where $\texttt{main}\,() = e_{\texttt{main}} \in d$. (Here, $()$ denotes the empty argument sequence.) We write $e \to_d^* e'$ for zero or more reductions from e to e'.

We assume that a program is typable with the standard simple type system[4] so that it is guaranteed to not get stuck, for example, by trying to use an integer as a function. Therefore, a program either runs forever safely (due to CPS, a program cannot return), or aborts with an assertion failure. We call a program *safe* if its run does not cause an assertion failure.

DEFINITION 2.1 (Safety). *A program d is said to be safe if $e_{\texttt{main}} \not\to_d^* \mathit{fail}$ where $\texttt{main}\,() = e_{\texttt{main}} \in d$.*

Being simply-typable does not imply safety. In the following, we present a dependent type system that guarantees the safety of typable programs.

2.1 Dependent Type System

Our dependent type system is essentially the previous systems [12, 34, 37] extended with intersection types. The reason for adding intersection types is not just to increase expressibility; it is actually crucial to the type inference system described later in the paper.

Recall that a program is simply-typed. For each expression e in the program, we write $\mathit{sty}(e)$ to denote its simple type. A simple type, s, is formally defined by the following grammar:

$$\begin{array}{rcl}B & ::= & \texttt{int} \mid \texttt{bool} \\ s & ::= & \star \mid B \mid s \to s'\end{array}$$

Here, B is called *base type*, and the dummy type $\star$ represents the type of a CPS expression.

Figure 5 shows the syntax of dependent types. Here, $\{u{:}B \mid \theta\}$ is a *refinement base type* that refines the base type B by the formula θ which is a formula in some first-order theory. We sometimes abbreviate $\{u{:}B \mid \theta\}$ simply as B when θ is a tautology (e.g., $\{u{:}\texttt{int} \mid \top\} = \texttt{int}$). Intuitively, $\{u{:}B \mid \theta\}$ denotes the type of some value u of the base type B satisfying the formula θ. The type $x : \sigma \to \tau$ is a *dependent function type* consisting of the argument type σ and the return type τ. Intuitively, $x{:}\sigma \to \tau$ denotes the type of a function (or a constant operation) that returns a value of the type $\tau[y/x]$ when applied to any argument y of the type σ.

The type $\{x{:}B \mid \theta\}$ binds x within θ. Likewise, $x : \sigma \to \tau$ binds x in τ (but not in σ). We sometimes abbreviate $x{:}\sigma \to \tau$ as $\sigma \to \tau$ when x does not occur free in τ. Types are equivalent up to renaming of bound variables.

We use the symbol σ to distinguish types with possible top-level intersections from those without (for which we use τ). Here, the intersection operator $\wedge$ is associative, commutative, and idempotent (ACI), so that, for example, $\tau \wedge \tau = \tau$. We sometimes write $\bigwedge_i \tau_i$ or $\bigwedge T$ for the type $\tau_1 \wedge \tau_2 \wedge \cdots \wedge \tau_n$ such that $\{\tau_1, \ldots, \tau_n\} = T$ is a non-empty set. Note that, because of ACI, any σ can be written in such a form.

For any intersection of types $\bigwedge_i \tau_i$, we enforce that each τ_i is of the same simple-type shape. Formally, the *simple-type shape* of σ is the simple type $\mathit{simple}(\sigma)$ defined inductively as follows:

$$\begin{array}{l}\mathit{simple}(x{:}\sigma \to \tau) = \mathit{simple}(\sigma) \to \mathit{simple}(\tau) \\ \mathit{simple}(\{u{:}B \mid \theta\}) = B \\ \mathit{simple}(\star) = \star \\ \mathit{simple}(\bigwedge\{\tau, \ldots\}) = \mathit{simple}(\tau)\end{array}$$

Then, we enforce that for any type $\bigwedge T$, $\mathit{simple}(\tau) = \mathit{simple}(\tau')$ for all $\tau, \tau' \in T$. This does not reduce expressibility because the type system is a refinement type system [14] of the simple type system, and so only the types of the same simple-type shape are meaningful to intersect. Without loss of generality, we implicitly assume that any dependent type σ assigned to e in the dependent type system satisfies $\mathit{simple}(\sigma) = \mathit{sty}(e)$.

Figure 6 shows the type checking rules of the dependent type system. The judgements are of the form $\Gamma;\theta \vdash e : \tau$ where Γ is a *type environment* mapping variables and function names to types possibly containing top-level intersections (i.e., σ's), and θ is a formula. The formula θ is used to accumulate the assumptions from nesting branch conditions.

We discuss each typing rule. **VaB** 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. The rule is borrowed from previous work [32, 34, 37]. **VaF** types function-type variables by looking up the environment and selecting a type from the intersection. Here, as usual, $\Gamma, x : \sigma$ denotes the mapping $\Gamma \cup \{x \mapsto \sigma\}$ if $x \notin \mathit{dom}(\Gamma)$, and is undefined otherwise. **Fun** is exactly like **VaF** except that it is for function names. **Cst** types constants. Here, $\mathit{ty}(c)$ is some sound[5] dependent type for the constant c (e.g., $\mathit{ty}(+) = x{:}\texttt{int} \to y{:}\texttt{int} \to \{u{:}\texttt{int} \mid u = x + y\}$).

Int1 and **Int2** introduce intersection types. Here, $\Gamma_1 \wedge \Gamma_2$ is defined as follows.

$$\begin{array}{rl}\Gamma_1 \wedge \Gamma_2 = & \{x \mapsto \Gamma_1(x) \wedge \Gamma_2(x) \mid x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2)\} \\ & \cup\{x \mapsto \Gamma_1(x) \mid x \in \mathit{dom}(\Gamma_1) \wedge x \notin \mathit{dom}(\Gamma_2)\} \\ & \cup\{x \mapsto \Gamma_2(x) \mid x \notin \mathit{dom}(\Gamma_1) \wedge x \in \mathit{dom}(\Gamma_2)\}\end{array}$$

We could have used a simpler set of rules that shares the environments of the sub-judgements because intersections are non-linear (recall that $\wedge$ is ACI), but this format makes the introduction of the linearity restriction smoother later in Section 5.2. Note that we write $\vdash_\wedge$ to distinguish judgements that can introduce top-level intersections.

[4] See Appendix A for the definition of the simple type system.

[5] See Appendix B for the definition of a sound constant type.

$$\frac{}{\Gamma;\theta \vdash \star \leq \star}\ \textbf{SubC}$$

$$\frac{u \notin \mathit{free}([\![\Gamma]\!]) \quad ([\![\Gamma]\!] \wedge \theta \wedge \theta_1) \Rightarrow \theta_2}{\Gamma;\theta \vdash \{u{:}B \mid \theta_1\} \leq \{u{:}B \mid \theta_2\}}\ \textbf{SubB}$$

$$\frac{\Gamma;\theta \vdash \sigma_2 \leq \sigma_1 \quad \Gamma, x:\sigma_2;\theta \vdash \tau_1 \leq \tau_2}{\Gamma;\theta \vdash x{:}\sigma_1 \to \tau_1 \leq x{:}\sigma_2 \to \tau_2}\ \textbf{SubF}$$

$$\frac{\exists i.(\Gamma;\theta \vdash \tau_i \leq \tau)}{\Gamma;\theta \vdash \bigwedge_i \tau_i \leq \tau}\ \textbf{SubI1} \qquad \frac{\forall i.(\Gamma;\theta \vdash \sigma \leq \tau_i)}{\Gamma;\theta \vdash \sigma \leq \bigwedge_i \tau_i}\ \textbf{SubI2}$$

Figure 7. The subtyping rules.

Let is self-explanatory. Note that the typing for e_1 may introduce top-level intersections. **App** types applications. Here, $\tau[x/y]$ is the usual capture-avoiding substitution. **App** checks that the actual argument conforms to the formal argument type via the subtyping $\Gamma_2;\theta \vdash \sigma' \leq \sigma$. Figure 7 shows the subtyping rules, which are a straightforward extension of those of the previous systems [12, 34, 37] with intersection types. In **SubB**, $[\![\Gamma]\!]$ is the first-order logic formula denoting the assumptions about the base-type variables, and is formally defined as follows.

$$[\![\Gamma]\!] = \bigwedge\{\bigwedge_i \theta_i[x/u] \mid \Gamma(x) = \bigwedge_i\{u{:}B \mid \theta_i\}\}$$

The **If** rule types conditional expressions. Note that the assumption about the branch condition (i.e., x) is recorded in the environment. Finally, **Assert** checks the assertion via subtyping.

We say that a type is *closed* if it has no free variables. Let Δ be a top-level type environment mapping function names to types. We say that σ is a *well-formed type* for F if σ is closed and $\mathit{simple}(\sigma) = \mathit{sty}(F)$. We say that Δ is a *well-formed top-level type environment* if $\Delta(F)$ is well-formed for each F. Unless mentioned otherwise, we restrict Δ to range only over well-formed top-level type environments in the rest of the paper.

Let us write $\overrightarrow{x{:}\sigma} \to \tau$ to abbreviate the function type

$$x_1{:}\sigma_1 \to \cdots \to x_n{:}\sigma_n \to \tau$$

where $\overrightarrow{x{:}\sigma} = x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n$. We define the notion of a well-typed program.

DEFINITION 2.2 (Well-typed program). *We write $\Delta \vdash d$ if for each function $F\ \vec{x} = e \in d$, we have $\Delta, \overrightarrow{x{:}\sigma}; \top \vdash e : \star$ for each $\tau_i = \overrightarrow{x{:}\sigma} \to \star$ in $\Delta(F) = \bigwedge_i \tau_i$.*

We say that a program d is well-typed (equivalently, typable) if there exists Δ such that $\Delta \vdash d$.

Assuming that the types of the constants $ty(c)$ are sound, the type system ensures that a well-typed program does not cause an assertion failure.

THEOREM 2.3 (Soundness). *If $\Delta \vdash d$ then d is safe.*

The proof is analogous to that of the soundness result for similar dependent type systems [12, 34, 37] and is omitted.

EXAMPLE 2.4. Let d consist of the following three functions. (We elide A-normalization for readability.)

```
sum x y k =
    if x ≤ 0 then k y else sum (x − 1) (x + y) k
check x =
    assert (100 ≤ x); check x
main () =
    sum 100 0 check
```

Note that `sum` is a function that, given integers x and y, computes $y + \sum_{i \in \{0,\ldots,x\}} i$ and applies the continuation k to the result. Therefore, d runs safely forever if $100 \leq \sum_{i \in \{0,\ldots,100\}} i$, and aborts with an assertion failure otherwise (i.e., it will run forever).

Assume that we are given the following constant types.

$$\begin{aligned}
&ty(\leq) = x{:}\mathtt{int} \to y{:}\mathtt{int} \to \{u{:}\mathtt{bool} \mid u = x \leq y\}\\
&ty(-) = x{:}\mathtt{int} \to y{:}\mathtt{int} \to \{u{:}\mathtt{int} \mid u = x - y\}\\
&ty(+) = x{:}\mathtt{int} \to y{:}\mathtt{int} \to \{u{:}\mathtt{int} \mid u = x + y\}\\
&ty(i) = \{u{:}\mathtt{int} \mid u = i\} \quad \text{where } i \in \{0, 1, 100\}
\end{aligned}$$

We show that we can prove d to be safe with our type system assuming that the underlying theory supports booleans and linear arithmetic.

Let Δ be the following typing environment.

$$\begin{aligned}
&\Delta(\mathtt{main}) = () \to \star\\
&\Delta(\mathtt{check}) = \{u{:}\mathtt{int} \mid u \geq 100\} \to \star\\
&\Delta(\mathtt{sum}) = \bigwedge T
\end{aligned}$$

where

$$\begin{aligned}
&T =\\
&\{\ x{:}\{u{:}\mathtt{int} \mid u \geq 100\} \to y{:}\{u{:}\mathtt{int} \mid u \geq 0\} \to\\
&\qquad\qquad (\{u{:}\mathtt{int} \mid u \geq 100\} \to \star) \to \star,\\
&\quad x{:}\mathtt{int} \to y{:}\mathtt{int} \to (\{u{:}\mathtt{int} \mid u \geq y\} \to \star) \to \star\ \}
\end{aligned}$$

(Here, $() \to \star$ is the special function type for `main` having an empty sequence of arguments. See Definition 2.2.)

It is a routine to check that $\Delta \vdash d$. Note that the type of `sum` does not say that it actually returns (i.e., calls the continuation with) $y + \sum_{i \in \{0,\ldots,x\}} i$, but only that it returns some integer at least as large as 100 when called with $x \geq 100$ and $y \geq 0$, and some integer at least as large as y unconditionally, which is sufficient for typing d.

3. Type Inference Overview

We now present our CEGAR-inspired procedure that checks if the given program d is typable, and if so, returns Δ such that $\Delta \vdash d$. The inference procedure is a semi-algorithm as it is not guaranteed to terminate, but it is sound and complete in that it is guaranteed to return some correct typing when terminating with success and reject the program as untypable only if it is actually untypable. (In practice, we make the procedure give up after some number of iterations, returning "unknown.")

The type inference maintains a lattice of candidate top-level type environments, and repeatedly executes the following two algorithms, one after the other.

- The *fixed-point type inference* algorithm checks if there exist a typing for the program within the current candidates via a fixed-point iteration over the lattice of candidate top-level type environments. The algorithm returns a counterexample if the program is found untypable with the current candidates. The counterexample records the number of times the fixed-point iteration was executed to reach the type error. Otherwise, the program is found typable and the process exits by returning the inferred typing. (Section 4)
- Given a counterexample, the *refinement* algorithm unwinds the recursive definitions the number of times recorded in the counterexample, generating a non-recursive program slice. Then, the algorithm decides the slice's typability completely (i.e., not restricted to any candidates). If the slice is found untypable, then so is the original, and the process exits. Otherwise, the dependent types that are used to type the slice are added as the new candidates to refine the lattice. This phase uses linear intersection type inference and interpolation to infer types for the unwound program slice. (Section 5)

The two components are both algorithms in that they are guaranteed to terminate. The following sections describe the two components in detail.

4. Fixed-point Type Inference

A *candidate set* Θ is a mapping from function symbols to a non-empty finite set of dependent types with no top-level intersections such that for all $\tau \in \Theta(F)$, τ is closed and $\mathit{simple}(\tau) = \mathit{sty}(F)$.

Θ induces the lattice $\bigwedge \Theta$ defined as follows.[6]

$$\bigwedge \Theta = \{\Delta \mid \forall F.\Delta(F) = \bigwedge T \text{ where } T \subseteq \Theta(F)\}$$

For $\Delta_1, \Delta_2 \in \bigwedge \Theta$, we order $\Delta_1 \preceq \Delta_2$ if for all F, we have $T_1 \supseteq T_2$ where $\Delta_1(F) = \bigwedge T_1$ and $\Delta_2(F) = \bigwedge T_2$. It is easy to see that $\bigwedge \Theta$ with $\preceq$ forms a lattice with the mapping $\lambda F. \bigwedge \emptyset$ as the top element and $\lambda F. \bigwedge \Theta(F)$ as the bottom element. Note that except for the ones containing $\bigwedge \emptyset$, any $\Delta \in \bigwedge \Theta$ is a well-formed top-level type environment.

We define the algorithm *InferNext* that takes $\Delta \in \bigwedge \Theta$ and returns the strongest typing $\Delta' \in \bigwedge \Theta$ for d that can be typed with Δ (d is an implicit parameter to *InferNext*). More precisely, for Δ a type environment (i.e., for all F, $\Delta(F) \neq \bigwedge \emptyset$), $\mathit{InferNext}(\Delta) = \Delta'$ such that for each $F\,\overrightarrow{x} = e \in d$,

$$\Delta'(F) = \bigwedge\{\overrightarrow{x{:}\sigma} \to \star \in \Theta(F) \mid \Delta, \overrightarrow{x{:}\sigma}; \top \vdash e : \star\}$$

If Δ is not a type environment, then we take $\mathit{InferNext}(\Delta) = \Delta$.

$\mathit{InferNext}(\Delta)$ can be effectively computed assuming that we can decide the typing judgements $\Gamma; \theta \vdash e : \tau$. The main complexity involved here is deciding the subtyping relation $\mathbf{SubB}$ (cf. Figure 7), which can be done by the help from a theorem prover supporting the underlying first order theory.

It is easy to see that if a fixed point of *InferNext* is a type environment then it is a valid typing for d. That is,

THEOREM 4.1. *Suppose $\Delta \in \bigwedge \Theta$ is a type environment such that $\mathit{InferNext}(\Delta) = \Delta$. Then, we have $\Delta \vdash d$.*

Moreover, it is easy to show that *InferNext* is monotonic. Then, the following theorem is immediate from the fact that $\bigwedge \Theta$ is a finite (and therefore, a complete) lattice.

THEOREM 4.2. *The least fixed point Δ of InferNext is a type environment if and only if there exists $\Delta' \in \bigwedge \Theta$ such that $\Delta' \vdash d$.*

Therefore, to decide if d is typable with the current candidate set, it suffices to compute the least fixed point $\Delta = \bigsqcup_{i \in \omega} \mathit{InferNext}^i(\lambda F. \bigwedge \Theta(F))$ and check that $\Delta(F) \neq \bigwedge \emptyset$ for all F. If such Δ exists, we stop the CEGAR process and return Δ as the inferred typing for d.

Otherwise, we have $\Delta = \mathit{InferNext}^i(\lambda F. \bigwedge \Theta(F))$ such that $\Delta(F) = \bigwedge \emptyset$ for some F at some iteration i. In this case, we pass the pair (F, i) as the *counterexample* to the refinement algorithm described in Section 5.

EXAMPLE 4.3. Let d consist of the following four functions.

$$\begin{array}{l} F\,x\,y = \texttt{if}\ x\ \texttt{then}\ F\,y\,x\ \texttt{else}\ G\,x\,y \\ G\,x\,y = \texttt{assert}\ y; F\,y\,x \\ H\,x = \texttt{assert}\ x; H\,x \\ \texttt{main}\,() = \texttt{if}\ \mathit{true}\ \texttt{then}\ F\,\mathit{true}\,\mathit{false}\ \texttt{else}\ H\,\mathit{false} \end{array}$$

Let the current candidate set be Θ shown below.

$$\begin{array}{l} \Theta(\texttt{main}) = \{() \to \star\} \\ \Theta(F) = \{\texttt{bool} \to \texttt{bool} \to \star, \\ \qquad \{u{:}\texttt{bool} \mid u = \mathit{true}\} \to \{u{:}\texttt{bool} \mid u = \mathit{false}\} \to \star\} \\ \Theta(G) = \{ \\ \qquad \{u{:}\texttt{bool} \mid u = \mathit{false}\} \to \{u{:}\texttt{bool} \mid u = \mathit{true}\} \to \star\} \\ \Theta(H) = \{\{u{:}\texttt{bool} \mid \bot\} \to \star\} \end{array}$$

[6] Here, Δ's are allowed to range over non type environments.

Let Δ_0 be the least element of the lattice $\bigwedge \Theta$, that is,

$$\begin{array}{l} \Delta_0 = \{\texttt{main} \mapsto \bigwedge \Theta(\texttt{main}), F \mapsto \bigwedge \Theta(F), \\ \qquad G \mapsto \bigwedge \Theta(G), H \mapsto \bigwedge \Theta(H)\} \end{array}$$

Then, $\mathit{InferNext}(\Delta_0) = \Delta_1$ where

$$\begin{array}{l} \Delta_1(\texttt{main}) = () \to \star \\ \Delta_1(F) = \{u{:}\texttt{bool} \mid u = \mathit{true}\} \to \{u{:}\texttt{bool} \mid u = \mathit{false}\} \to \star \\ \Delta_1(G) = \{u{:}\texttt{bool} \mid u = \mathit{false}\} \to \{u{:}\texttt{bool} \mid u = \mathit{true}\} \to \star \\ \Delta_1(H) = \{u{:}\texttt{bool} \mid \bot\} \to \star \end{array}$$

Δ_1 is a type environment but $\Delta_1 \neq \Delta_0$. Therefore, iterating one more time, we get $\mathit{InferNext}(\Delta_1) = \Delta_2$ where $\Delta_2(\texttt{main}) = \Delta_1(\texttt{main})$, $\Delta_2(G) = \Delta_1(G)$, $\Delta_2(H) = \Delta_1(H)$, but $\Delta_2(F) = \bigwedge \emptyset$. Because $\Delta_2(F)$ is not a type, we have reached the fixed point and can return $(F, 2)$ as the counterexample.

5. Refinement

Recall that the goal of the refinement phase is to check if the counterexample is spurious by constructing a non-recursive program fragment from the counterexample and checking if the fragment is typable, not restricted to any candidate set. And if so, we build new candidates from the inferred typing, and otherwise, we can reject the program as untypable.

We separate the refinement phase into the following three sub-phases, executed in order.

- We take the counterexample (F, i) and *unwind* the recursive definitions i times from F to produce a non-recursive program fragment. (Section 5.1)
- We infer the type derivation shape that is sufficient for typing the unwound fragment via linear intersection type inference. (Section 5.2)
- Given the linear derivation shape, we generate constraints consisting of first-order logic formulas and predicate variables, and check if the constraints are satisfiable by using an interpolating theorem prover. If not, then the unwound fragment is untypable, and we stop the CEGAR process declaring the program untypable. Otherwise, from the typing inferred for the fragment, we produce new candidate types that are sufficient to refute the counterexample. (Sections 5.3 and 5.4)

Next, we describe the three sub-phases in detail.

5.1 Unwinding

The unwinding phase is the simplest phase of the refinement process. Given a counterexample (F, i), we inline recursive definitions i times from F in d, leaving *leaf* function name occurrences with no definitions. The resulting program fragment, d', is then guaranteed to be free of recursive definitions.

We demonstrate the process by unwinding the program d from Example 4.3. Recall that we are given the counterexample $(F, 2)$. Then, unwinding produces the following program slice d':

$$\begin{array}{l} G_1\,x\,y = \texttt{assert}\ y; F_4\,y\,x \\ F_2\,x\,y = \texttt{if}\ x\ \texttt{then}\ F_3\,y\,x\ \texttt{else}\ G_2\,x\,y \\ F_1\,x\,y = \texttt{if}\ x\ \texttt{then}\ F_2\,y\,x\ \texttt{else}\ G_1\,x\,y \end{array}$$

Here, $F_1, F_2, F_3, F_4, G_1, G_2$ are inlined function names created fresh. Note that F_3, F_4, and G_2 are leaf functions. We maintain the mapping *Inames* that maps the original function names of d to the set of its inlined copies in d'. In the above example, $\mathit{Inames}(F) = \{F_1, F_2, F_3, F_4\}$, $\mathit{Inames}(G) = \{G_1, G_2\}$, and $\mathit{Inames}(H) = \mathit{Inames}(\texttt{main}) = \emptyset$. Note that the unwinding only contains functions that are involved in the counterexample, that is, `main` and H (as well as F and G beyond depth 2) are sliced out.[7]

We discuss the key properties of unwinding. First, because d is simply typable, d' is also simply typable, and we assume that $sty(e)$ is given for each expression e in d'. Let us extend the type judgement $\vdash$ to leaf function name occurrences in the obvious way by allowing any well-formed type for the function to be assigned. The following is immediate.

THEOREM 5.1. *Suppose $\Delta \vdash d$ and d' is an unwinding of d. Let Δ' be a top-level type environment for d' such that for each $G \in \mathit{Inames}(F)$, $\Delta'(G) = \Delta(F)$. Then, $\Delta' \vdash d'$.*

Therefore, showing that d' is untypable is sufficient for showing that d is untypable.

As a contrapositive, we show that the current candidate set is insufficient for typing d', using candidates for the originals for the inlined functions.

THEOREM 5.2. *Let d' be the unwinding of d produced from the counterexample passed from the fixed-point type inference phase using the candidate sets Θ. Let Θ' be a candidate set for d' such that for each $G \in \mathit{Inames}(F)$, $\Theta'(G) = \Theta(F)$. Then, there exists no $\Delta \in \bigwedge \Theta'$ such that $\Delta \vdash d'$.*

The above theorems justify us calling (F, i) a *counterexample*. That is, the unwinding d' produced from (F, i) is a counterexample to the typability of d under the current candidate set.

5.2 Linear Intersection Types

The goal of the rest of the refinement phase is to check if the unwinding d' is typable, without confining the types of the functions to any candidate set. An issue here is that the type system allows unboundedly many intersections, and so we cannot naively derive a type inference algorithm from the type checking rules from Section 2.1. To overcome the issue, we use the observation that only *linear* intersections are needed for typing d' and that the linear intersection "shapes" can be inferred. The crucial properties of d' that enable this is that d' does not contain recursive definitions and is simply typable. Linearity is also important to the constraint solving phase of the refinement algorithm because it ensures the acyclicity of the generated constraints (cf. Section 5.3).

Informally, in the *linear intersection dependent type system* $\vdash^1$, for a non-base-type binding $x : \sigma$, the top-level intersections of σ determine how the variable x is used. For example, it is possible to derive

$$\Gamma_2; \top \vdash^1 \texttt{if}\ y\ \texttt{then}\ x\,y\ \texttt{else}\ x\,y : \star$$

where $\Gamma_2 = y : \texttt{bool}, x{:}(\texttt{bool} \rightarrow \star) \wedge (\texttt{bool} \rightarrow \star)$, but

$$\Gamma_1; \top \not\vdash^1 \texttt{if}\ y\ \texttt{then}\ x\,y\ \texttt{else}\ x\,y : \star$$
$$\Gamma_3; \top \not\vdash^1 \texttt{if}\ y\ \texttt{then}\ x\,y\ \texttt{else}\ x\,y : \star$$

where $\Gamma_1 = y : \texttt{bool}, x : \texttt{bool} \rightarrow \star$ and $\Gamma_3 = y : \texttt{bool}, x : (\texttt{bool} \rightarrow \star) \wedge (\texttt{bool} \rightarrow \star) \wedge (\texttt{bool} \rightarrow \star)$.

Essentially, $\vdash^1$ is equivalent to $\vdash$ except that it disallows non-linear use of function-type bindings. The linearity restriction is only imposed on function types; base types are used non-linearly.

We formally define the type system $\vdash^1$. The syntax of linear intersection types is equivalent to that of $\vdash$ (cf. Figure 5). But now, $\wedge$ is neither associative, commutative, nor idempotent. We also modify the typing rules so that the rules in Figure 8 replace **VaB**, **VaF**, **Fun**, **Cst** and **Let** from Figure 6 and **SubI1** and **SubI2** from Figure 7. The rest of the rules remain the same, just replacing $\vdash$ with $\vdash^1$, and $\vdash_\wedge$ with $\vdash^1_\wedge$. We eliminate any intersection of base types via the equivalence $\{u{:}B \mid \theta_1\} \wedge \{u{:}B \mid \theta_2\} = \{u{:}B \mid \theta_1 \wedge \theta_2\}$. As with $\vdash$, we assume that any type σ assigned to an expression e satisfies $\mathit{simple}(\sigma) = \mathit{sty}(e)$.

$$\frac{\mathit{sty}(x)\ \text{is base} \quad \mathit{isBaseEnv}(\Gamma)}{\Gamma; \theta \vdash^1 x : \{u{:}B \mid u = x\}}\ \mathbf{VaB^1}$$

$$\frac{\mathit{sty}(x)\ \text{is not base} \quad \mathit{isBaseEnv}(\Gamma)}{\Gamma, x{:}\tau; \theta \vdash^1 x : \tau}\ \mathbf{VaF^1}$$

$$\frac{\mathit{isBaseEnv}(\Gamma)}{\Gamma; \theta \vdash c : \mathit{ty}(c)}\ \mathbf{Cst^1} \qquad \frac{F\ \text{is leaf} \quad \mathit{isBaseEnv}(\Gamma)}{\Gamma, F{:}\tau; \theta \vdash^1 F : \tau}\ \mathbf{Fun1^1}$$

$$\frac{\begin{array}{c}\mathit{dom}(\Gamma')\ \text{are all function names} \quad \mathit{isBaseEnv}(\Gamma)\\ F\ \overrightarrow{x} = e \in d' \qquad \Gamma', \overrightarrow{x{:}\sigma} \setminus X; \top \vdash^1 e : \star\\ X = \{x_i \in \{\overrightarrow{x}\} \setminus \mathit{free}(e) \mid \mathit{sty}(x_i)\ \text{not base}\}\end{array}}{\Gamma, \Gamma', F{:}\overrightarrow{x{:}\sigma} \rightarrow \star; \theta \vdash^1 F : \overrightarrow{x{:}\sigma} \rightarrow \star}\ \mathbf{Fun2^1}$$

$$\frac{\begin{array}{c}\Gamma_1; \theta \vdash^1_\wedge e_1 : \sigma \quad \Gamma_2, x{:}\sigma \setminus X; \theta \vdash^1 e_2 : \star\\ X = \{x \mid x \notin \mathit{free}(e_2)\ \text{and}\ \mathit{sty}(x)\ \text{not base}\}\end{array}}{\Gamma_1 \wedge \Gamma_2; \theta \vdash^1 \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \star}\ \mathbf{Let^1}$$

$$\frac{\Gamma \vdash^1 \sigma_1 \leq \sigma_1' \quad \Gamma \vdash^1 \sigma_2 \leq \sigma_2'}{\Gamma \vdash^1 \sigma_1 \wedge \sigma_2 \leq \sigma_1' \wedge \sigma_2'}\ \mathbf{SubI^1}$$

Figure 8. The key $\vdash^1$ typing rules.

We discuss the new rules. $\mathbf{VaB^1}$ replaces **VaB**. Here, the condition $\mathit{isBaseEnv}(\Gamma)$ says that all bindings in Γ are base types, that is, $\mathit{dom}(\Gamma)$ does not contain function names and for all $x \in \mathit{dom}(\Gamma)$, $\Gamma(x)$ is a base type. $\mathbf{VaF^1}$ replaces **VaF** and requires that the only function-type binding in the environment is x. Similarly, $\mathbf{Cst^1}$ replaces **Cst** and requires that Γ contains no function-type bindings.

$\mathbf{Fun1^1}$ and $\mathbf{Fun2^1}$ replace **Fun**. $\mathbf{Fun1^1}$ types leaf function names and is much like $\mathbf{VaF^1}$. $\mathbf{Fun2^1}$ types non-leaf function names. (Note that judgements are implicitly parameterized by the unwound fragment d'.) Unlike **Fun**, it type checks the body of the function in the sub-derivation. Note that this does not lead to an infinite derivation tree because d' does not contain recursive definitions. Here, $\overrightarrow{x{:}\sigma} \setminus X$ denotes the bindings $\overrightarrow{x{:}\sigma}$ with the bindings for X removed. We need this because linearity implies that a non-base-type variable that are not used in e cannot be bound in the environment. $\mathbf{Let^1}$ similarly takes care of unused non-base variable bindings. Finally, $\mathbf{SubI^1}$ replaces **SubI1** and **SubI2** from Figure 7. It just structurally applies subtyping inside intersections.

Note that any unwound program fragment has a unique *root* function from where the unwinding started and whose name does not occur free in the unwinding. We define the notion of linearly typable programs.

DEFINITION 5.3 (Linearly typable programs). *Let d' be an unwound program fragment with F as the root function. Then, we write $\Delta \vdash^1 d'$ if $\Delta \vdash^1 F : \tau$ for some τ. (Incidentally, it must be the case that $\tau = \Delta(F)$.)*

The following theorem states that we can decide if d' is typable by deciding if d' is linearly typable.

THEOREM 5.4. *Let d' be an unwound program fragment. Then, the following are equivalent.*

(1) There exists Δ such that $\Delta \vdash^1 d'$.
(2) There exists Δ such that $\Delta \vdash d'$.

[7] The implementation performs additional optimizations that can further reduce the size of the unwinding by slicing out more irrelevant parts.

We defer the proof to Appendix C.

Unlike $\vdash$, $\vdash^1$ is completely structural because the shape of a derivation, including the number of intersections in the types, is determined by how variables occur in d'. To infer the derivation shape, we adopt the *expansion-variable*-based inference algorithm of Kfoury and Wells [21], modified so that expansions are not applied to base-type bindings. For space, we refer to their paper [21] for the details of the algorithm. The inferred shape satisfies all the structural requirements of $\vdash^1$, that is, everything except for the logical validity premise at $\mathbf{SubB}$.

More precisely, we introduce *shape-only types* that have *holes* "$-$" in place of first-order logic formulas, defined as follows.

$$\begin{array}{rcl} \hat{\tau} & ::= & \star \mid \{x{:}B \mid -\} \mid x{:}\hat{\sigma} \to \hat{\tau} \\ \hat{\sigma} & ::= & \hat{\tau} \mid \hat{\sigma}_1 \wedge \hat{\sigma}_2 \end{array}$$

Let *shape-only type environment* $\hat{\Gamma}$ be a mapping from variables and function names to $\hat{\sigma}$'s, and *top-level shape-only type environment* $\hat{\Delta}$ be a mapping from functions names to $\hat{\sigma}$'s. Then, *linear intersection type shape inference judgement* $\hat{\Gamma}; -\hat{\vdash}^1 e : \hat{\tau}$ consists of $\vdash^1$ rules, but using $\hat{\tau}$ (resp. $\hat{\sigma}$) wherever τ (resp. σ) appears, using the hole $-$ for formulas, and replacing $\mathbf{SubB}$ with the following rule.

$$\frac{}{\hat{\Gamma}; -\hat{\vdash}^1 \{u{:}B \mid -\} \leq \{u{:}B \mid -\}}$$

We also replace formulas in constant types $ty(c)$ with holes, prohibit intersection of base types, and use types containing no intersections for the unused bindings X at $\mathbf{Fun2^1}$ and $\mathbf{Let^1}$. Then, the *linear intersection type shape inference* infers a shape-only derivation $\hat{\Delta}\hat{\vdash}^1 d'$. The fact that such a derivation exists is the consequence of the fact that d' is recursion free and is simply typable,[8] and follows from well-known properties of intersection types (see, e.g., [21]).

The linear intersection type shape inference is non-trivial in the presence of higher-order functions. In fact, it is known that the complexity of the inference is non-elementary time hard [30, 35]. But the expansion-variable-based algorithm appears to work well in practice, perhaps because functions of high ranks,[9] which could cause the inference to explode, are used sparingly in practice.

5.3 Constraint Generation and Constraint Solving

Having generated the derivation shape for d', the next step is to check if the shape can be turned into an actual $\vdash^1$ derivation by filling in the holes with first-order logic formulas. To this end, we introduce *predicate variables* that serve as placeholders for first-order logic formulas in the derivation. We use large letters P, Q, etc. to range over predicate variables. We generate constraints containing predicate variables and formulas from the underlying theory, and use an interpolating theorem prover to solve for the predicate variables.

We extend the syntax of dependent types to allow predicate variables in place of formulas in refinement base types:

$$\begin{array}{rcl} \rho & ::= & \varepsilon \mid \rho, [x/y] \\ \tau & ::= & \cdots \mid \{u{:}B \mid P\rho\} \end{array}$$

(We overload τ to range over the extended types when it is clear from the context.) The sequence of substitutions ρ is called *pending substitutions* [34] (or *delayed substitutions* [23]). Pending substitutions records the substitutions $\tau[x/y]$ made at $\mathbf{App}$ so that, for example, $\{u{:}B \mid P\}[x/y] = \{u{:}B \mid P[x/y]\}$. (The empty substitution ε is elided.)

[8] Actually, typability under any type system that ensures normalization of recursion-free terms is sufficient.

[9] Roughly, a *rank* is the number of times a type can be nested in the left hand side of $\to$.

Let $\hat{\Delta}\hat{\vdash}^1 d'$ be the inferred derivation shape. $\hat{\Delta}$ is a mapping from non-root functions in d' to shape-only types. We build a *function type template* $\ddot{\Delta}$ from $\hat{\Delta}$ such that for each $\hat{\Delta}(F) = \hat{\sigma}$, we have $\ddot{\Delta}(F) = \sigma$ where σ is $\hat{\sigma}$ with its holes filled with fresh predicate variables with empty pending substitutions. For example, from the shape

$$\hat{\Delta} = \{F \mapsto x{:}\{u{:}\mathtt{int} \mid -\} \to \star, G \mapsto x{:}\{u{:}\mathtt{int} \mid -\} \to \star\}$$

we make the template

$$\ddot{\Delta} = \{F \mapsto x{:}\{u{:}\mathtt{int} \mid P\} \to \star, G \mapsto x{:}\{u{:}\mathtt{int} \mid Q\} \to \star\}$$

where $P \neq Q$. (Here, we write $\ddot{\Delta}$ to emphasize that it may contain predicate variables.)

To generate constraints, we convert $\vdash^1$ type checking rules to constraint generation rules by modifying the base-type subtyping rule $\mathbf{SubB}$ so that instead of checking logical validity, the premise $[\![\Gamma]\!] \wedge \ddot{\theta} \wedge \ddot{\theta_1} \Rightarrow \ddot{\theta_2}$ is recorded as a constraint. Here, we use the symbol $\ddot{\theta}$ to range over formulas possibly containing predicate variables, (and reserve θ for *concrete* formulas that do not contain predicate variables). Then, having the template $\ddot{\Delta}$ at hand, we follow the derivation shape and record the constraints that occur at each $\mathbf{SubB}$ instance. Let $\mathcal{C}$ be the set of constraints obtained this way. Note that $\mathcal{C}$ is a set of formulas containing predicate variables.

To define a solution for the constraints, we define the *scope variables* of each P in the template $\ddot{\Delta}$, written *scopevars*(P), to be the set of variables that are allowed to appear free in a solution. Formally, *scopevars* is a mapping from predicate variables to the largest set of variables such that for any mapping S from predicate variables to concrete formulas with $\mathit{free}(S(P)) \subseteq \mathit{scopevars}(P)$ for all P, $S(\sigma)$ is closed for all $\sigma \in \mathit{ran}(\ddot{\Delta})$. Here, $S(\sigma)$ denotes σ with its predicate variables P replaced by the concrete formula $S(P)$. Scope variables can be computed by a linear scan over the template. For example, $\mathit{scopevars}(P) = \{x, u\}$ for $\ddot{\Delta} = \{F \mapsto x{:}\sigma \to y{:}\{u{:}\mathtt{int} \mid P\} \to \star\}$.

We say that S is a *solution* for $\mathcal{C}$, written $S \models \mathcal{C}$, if $S(P) \subseteq \mathit{scopevars}(P)$ for all P and for each $\ddot{\theta} \in \mathcal{C}$, $S(\ddot{\theta})$ is valid. It is easy to see that if S is a solution, then $S(\ddot{\Delta}) \vdash^1 d'$ where $S(\ddot{\Delta})$ is defined $\{F \mapsto S(\ddot{\Delta}(F)) \mid F \in \mathit{dom}(\ddot{\Delta})\}$.

5.3.1 Least Solution

To solve the constraints, we first compute the least solutions for the predicate variables. The least solutions are used when computing interpolants in the second phase of the constraint solving (cf. Section 5.3.2).

Note that each constraint in the generated set of constraints $\mathcal{C}$ is either of the following forms. (Recall that $\Rightarrow$ binds the weakest.)

(1) $\Psi \wedge \theta \Rightarrow \theta'$

(2) $\Psi \wedge \theta \Rightarrow P\rho$

where θ and θ' do not contain predicate variables and Ψ is a conjunction of predicate variables with pending substitutions:

$$\Psi ::= \top \mid \Psi \wedge P\rho$$

Predicate variables appearing in a constraint are guaranteed to be distinct. Furthermore, $\mathcal{C}$ is acyclic in the following sense.

THEOREM 5.5. *The generated set of constraints $\mathcal{C}$ does not contain constraints of the form $\{P_i\rho_i \wedge \ddot{\theta}_i \Rightarrow P_{i+1}\rho'_i \mid 1 \leq i \leq n\}$ for some $n \geq 1$ with $P_1 = P_{n+1}$.*

We defer the proof to Appendix C. The result follows from the fact that the derivation is linear. Indeed, cyclic constraints could be generated if we had used the simple types to obtain the derivation shape as seen Example 5.6 below.

Because of acyclicity, we can totally order predicate variables via topological sort so that if there is a constraint of the form $P\rho \wedge \ddot{\theta} \Rightarrow Q\rho'$ in $\mathcal{C}$, then $P < Q$.

EXAMPLE 5.6. This example illustrates the importance of linearity in avoiding cyclic constraints. Let the unwound fragment d' consist of the following functions.[10]

$$\begin{array}{lll} H\,f\,k = f\,0\,(G\,f\,k) & G\,f\,k\,x = f\,x\,k & F\,x\,k = k\,x \\ K\,x = \texttt{assert}\;x \leq 0; Kx & \texttt{main}\,() = H\;F\;K & \end{array}$$

Let $\hat{\Delta} \hat{\vdash}^1 d'$ be the inferred linear intersection type shape. Then,

$$\hat{\Delta}(H) = f{:}\hat{\tau} \wedge \hat{\tau} \rightarrow k{:}(x{:}\{u{:}\texttt{int} \mid -\} \rightarrow \star) \rightarrow \star$$

where $\hat{\tau} = \{u{:}\texttt{int} \mid -\} \rightarrow (\{u{:}\texttt{int} \mid -\} \rightarrow \star) \rightarrow \star$. From $\hat{\Delta}$, we get the template $\ddot{\Delta}$ such that $\ddot{\Delta}(H) = f{:}\tau_1 \wedge \tau_2 \rightarrow \ldots$ where $\tau_1 = \{u{:}\texttt{int} \mid P_1\} \rightarrow \ldots$ and $\tau_2 = \{u{:}\texttt{int} \mid P_2\} \rightarrow \ldots$, for P_1 and P_2 fresh. The set of constraints $\mathcal{C}$ generated from $\ddot{\Delta} \vdash^1 d'$ is acyclic, but induces the ordering $P_1 < P_2$. Hence, if we had used simple-type shapes so that $\hat{\Delta}(H) = f{:}\hat{\tau} \rightarrow \ldots$ instead, then we would have $P_1 = P_2$, and the constraints become cyclic. □

Thanks to acyclicity, we can systematically derive the least solution for $\mathcal{C}$ in a bottom up manner, that is, in the ascending order of $<$. (Although we call it the least *solution*, it is an actual solution only if $\mathcal{C}$ is satisfiable.)

We describe the method to obtain the least solution for P having obtained the least solutions for all $Q < P$. Let

$$\{\ddot{\theta}_1 \Rightarrow P\rho_1, \ddot{\theta}_2 \Rightarrow P\rho_2, \ldots, \ddot{\theta}_n \Rightarrow P\rho_n\}$$

be the set of constraints in $\mathcal{C}$ of the form $\ddot{\theta} \Rightarrow P\rho$. We set the least solution for P to be

$$\mathit{Least}(P) = \exists X. \bigvee_i \mathit{Least}(\ddot{\theta}_i)\rho_i^{-1}$$

where $X = \mathit{free}(\bigvee_i \mathit{Least}(\ddot{\theta}_i)\rho_i^{-1}) \setminus \mathit{scopevars}(P)$. We explain the construction. We first concretize each lowerbound $\ddot{\theta}_i$ of $P\rho_i$ by substituting the least solutions for the predicate variables appearing free in $\ddot{\theta}_i$. Note that such least solutions are already obtained. Then, we reverse substitute the pending substitutions ρ_i to obtain the concretized lowerbounds for P, that is, $\mathit{Least}(\ddot{\theta}_i)\rho_i^{-1}$ for each i. The correctness of the construction requires the targets of ρ_i to not occur free in $\mathit{Least}(\ddot{\theta}_i)$, which can be met by renaming the bound variables in the types. Finally, we take the disjunction of the concretized lowerbounds, and existentially quantify all free variables except for the scope variables of P. The latter ensures that $\mathit{free}(\mathit{Least}(P)) \subseteq \mathit{scopevars}(P)$.

The following is immediate from the construction of Least.

THEOREM 5.7. *If $\mathcal{C}$ is satisfiable, then Least is the least solution for $\mathcal{C}$. That is, Least $\models \mathcal{C}$, and for all solutions S such that $S \models \mathcal{C}$, $\mathit{Least}(P) \Rightarrow S(P)$ for all P.*

Thus, to check $\mathcal{C}$'s satisfiability, it suffices to check whether $\mathit{Least}(\ddot{\theta})$ is logically valid for each constraint $\ddot{\theta} \in \mathcal{C}$. And if so, we have $\mathit{Least}(\ddot{\Delta}) \vdash^1 d'$, and we can obtain new candidate types from $\mathit{Least}(\ddot{\Delta})$. (Recall that $\ddot{\Delta}$ is the template obtained from $\hat{\Delta}$ such that $\hat{\Delta}\hat{\vdash}^1 d'$.) While this is a sound and complete way to solve $\mathcal{C}$ and generate new candidates, it tends to produce suboptimal candidates.

There are two problems with using $\mathit{Least}(\ddot{\Delta})$ as the new candidates. One problem is that the least solutions may contain existential quantifiers that need to be eliminated before using them as candidates so that the fixed point type inference phase only needs to work with quantifier-free formulas.

[10] The code is somewhat contrived because of the CPS restriction. The essence is F being applied to the "return value" of another instance of itself.

```
sum1 x y k =
   if x ≤ 0 then k y else sum2 (x − 1) (x + y) k
check1 x =
   assert (100 ≤ x); check2 x
main1 () =
   sum1 100 0 check1
```

Figure 9. Example 2.4 unwound twice from `main`.

The second, more critical, issue is that the least solutions are often too strong as candidates. We illustrate the problem using the summation program d from Example 2.4. Suppose we are given the counterexample $(\texttt{main}, 2)$. Unwinding d twice from `main`, we obtain d' shown in Figure 9. Then, using the least solutions, we obtain the following top-level type environment $\Delta = \mathit{Least}(\ddot{\Delta})$.

$$\begin{array}{l} \Delta(\texttt{sum2}) = \\ \quad x{:}\{u{:}\texttt{int} \mid u = 99\} \rightarrow y{:}\{u{:}\texttt{int} \mid u = 100\} \rightarrow \\ \hfill (\{u{:}\texttt{int} \mid \bot\} \rightarrow \star) \rightarrow \star \\ \Delta(\texttt{sum1}) = \\ \quad x{:}\{u{:}\texttt{int} \mid u = 100\} \rightarrow y{:}\{u{:}\texttt{int} \mid u = 0\} \rightarrow \\ \qquad (\{u{:}\texttt{int} \mid \bot\} \rightarrow \star) \wedge (\{u{:}\texttt{int} \mid \bot\} \rightarrow \star) \rightarrow \star \\ \Delta(\texttt{check1}) = (\{u{:}\texttt{int} \mid \bot\} \rightarrow \star) \wedge (\{u{:}\texttt{int} \mid \bot\} \rightarrow \star) \\ \Delta(\texttt{check2}) = \{u{:}\texttt{int} \mid \bot\} \rightarrow \star \end{array}$$

Note that Δ says that `sum2` (resp. `sum1`) can only take 99 (resp. 100) as its first argument. Nevertheless, we have $\Delta \vdash^1 d'$, and so Δ is sufficient for typing d'. However, it is too strong to type the original program. Indeed, if we had used the least solutions to build candidates every time, then we would be generating a hundred candidate types to type the original program.

More generally, the least solutions are too strong when the unwinding is "incomplete," which is often the case for programs containing recursive functions. We would suffer from the dual problem had we used the greatest solutions instead, that is, they would be too weak. To overcome these issues, we use interpolation [11] to find a quantifier-free solution Interp that can be weaker than Least (i.e., $\mathit{Least}(P) \Rightarrow \mathit{Interp}(P)$) but is still strong enough to type d' (i.e., $\mathit{Interp} \models \mathcal{C}$).

5.3.2 Interpolants

We compute Interp via interpolation and the least solution Least obtained by the process in Section 5.3.1. We briefly review the basic properties of interpolation.

Interpolation Review Given formulas θ_1 and θ_1 where $\theta_1 \Rightarrow \theta_2$, an *interpolant* between θ_1 and θ_2, written $\langle\theta_1, \theta_2\rangle$, is a formula θ such that $\theta_1 \Rightarrow \theta$, $\theta \Rightarrow \theta_2$, and $\mathit{free}(\theta) \subseteq \mathit{free}(\theta_1) \cap \mathit{free}(\theta_2)$. It is known that quantifier-free interpolants exist and can be computed for many useful first-order theories, such as the quantifier-free theory of linear arithmetic and uninterpreted function symbols [5, 20, 27].

Recall the total ordering of predicate variables $<$ from Section 5.3.1. We compute $\mathit{Interp}(P)$ for each P in the descending order of $<$. We describe how to compute $\mathit{Interp}(P)$ having computed $\mathit{Interp}(Q)$ for all $Q > P$. Let

$$\{P\rho_1 \wedge \ddot{\theta}_1 \Rightarrow \ddot{\theta}'_1, P\rho_2 \wedge \ddot{\theta}_2 \Rightarrow \ddot{\theta}'_2, \ldots, P\rho_n \wedge \ddot{\theta}_n \Rightarrow \ddot{\theta}'_n\}$$

be the set of constraints in $\mathcal{C}$ of the form $P\rho \wedge \ddot{\theta} \Rightarrow \ddot{\theta}'$. We set $\mathit{Interp}(P) = \langle \mathit{Least}(P), \theta\rangle$ if $\mathit{Least}(P) \Rightarrow \theta$ where

$$\theta = \bigwedge_i S(\ddot{\theta}_i \Rightarrow \ddot{\theta}'_i)\rho_i^{-1}$$

where S is the substitution such that $S(Q) = \mathit{Interp}(Q)$ if $P < Q$ and $S(Q) = \mathit{Least}(Q)$ if $Q < P$. (P is guaranteed to not appear in $\ddot{\theta}_i \Rightarrow \ddot{\theta}'_i$.) Otherwise, $\mathit{Least}(P) \not\Rightarrow \theta$ and we reject d', and therefore also the original program d, as untypable.

We explain the construction of the upperbound θ above. First, we concretize the upperbound of each $P\rho_i$ (i.e., $\ddot{\theta}_i \Rightarrow \ddot{\theta}'_i$) by substituting formulas for the free predicate variables via S. Some free predicate variables in $\ddot{\theta}_i \Rightarrow \ddot{\theta}'_i$ may not have its Interp computed yet, and so S uses Least for such predicate variables. However, because of acyclicity, such predicate variables are guaranteed to appear only negatively (i.e., to the left of $\Rightarrow$). Then, we reverse substitute ρ_i to obtain the concretized upperbounds for P, that is, $S(\ddot{\theta}_i \Rightarrow \ddot{\theta}'_i)\rho_i^{-1}$ for each i. The correctness of the construction requires that the targets of ρ_i do not occur free in $S(\ddot{\theta}_i \Rightarrow \ddot{\theta}'_i)$, which can again be ensured by renaming of bound variables. Finally, we take the conjunction of the concretized upperbounds to obtain θ.

The following theorem states that the above algorithm finds a correct Interp if and only if $\mathcal{C}$ is satisfiable.

THEOREM 5.8. *The algorithm computes Interp such that Interp $\models$ $\mathcal{C}$ if and only if $\mathcal{C}$ is satisfiable.*

The proof is by induction on the (totally-ordered) predicate variables. See Appendix C for details. Note that because $\mathit{free}(\mathit{Least}(P)) \subseteq \mathit{scopevars}(P)$, we have $\mathit{free}(\mathit{Interp}(P)) \subseteq \mathit{scopevars}(P)$ from the property of interpolants.

From the construction, we have $\mathit{Least}(P) \Rightarrow \mathit{Interp}(P)$ for all P. Although Theorem 5.8 does not imply that $\mathit{Interp} \neq \mathit{Least}$, interpolating theorem provers tend to produce small interpolants that often work better as candidates than do the least solutions.

Computing without Quantifiers It is possible to compute the interpolant $\langle \mathit{Least}(P), \theta\rangle$ by renaming the existentially quantified variables in the least solutions with fresh variables. This approach is justified by the following lemma and the fact that existential quantifiers appear only positively in $\mathit{Least}(P)$ and only negatively in θ.

LEMMA 5.9. *A formula θ is an interpolant between $\exists X.\theta_1$ and $(\exists Y.\theta_2) \Rightarrow \theta_3$ if and only if it is an interpolant between $\theta_1[\overrightarrow{x}/X]$ and $\theta_2[\overrightarrow{y}/Y] \Rightarrow \theta_3$ where $\overrightarrow{x}$ and $\overrightarrow{y}$ are fresh variables.*

Thus, we can check for the satisfiability of $\mathcal{C}$ and obtain a quantifier-free Interp without explicit quantifier elimination by using a theorem prover that can produce quantifier-free interpolants for the formulas in the underlying theory.

5.4 New Candidate Types

Given Interp, we generate the new candidates sufficient for typing the counterexample. Recall that we have $\mathit{Interp}(\ddot{\Delta}) \vdash^1 d'$. $\mathit{Interp}(\ddot{\Delta})$ is a mapping from function names in d' to their types. Recall that *Inames* maps the original function names in d to their inlined copies in d'.

We define the new candidates Θ' as follows: for each F in d, $\Theta'(F) = \{\tau_i \mid \bigwedge_i \tau_i = \mathit{Interp}(\ddot{\Delta})(G)$ where $G \in \mathit{Inames}(F)\}$. Then, we pass Θ' to the fixed-point type inference component which updates the candidates by $\Theta := \Theta \uplus \Theta'$, where $\Theta \uplus \Theta'$ is the point-wise union $\lambda F.\Theta(F) \cup \Theta'(F)$. From the property $\mathit{free}(\mathit{Interp}(P)) \subseteq \mathit{scopevars}(P)$, the new types are guaranteed to be closed, and so the updated Θ is a valid candidate set. Furthermore, because $\mathit{Interp}(\ddot{\Delta}) \vdash d'$, it follows from Theorem 5.2 that the updated candidate set is sufficient to eliminate the spurious counterexample from future CEGAR iterations, as stated in the following theorem.

THEOREM 5.10. *Suppose that the lattice of candidates was refined by refuting the counterexample (F, i). Then, for any counterexample (F, j) reported in future, $j > i$.*

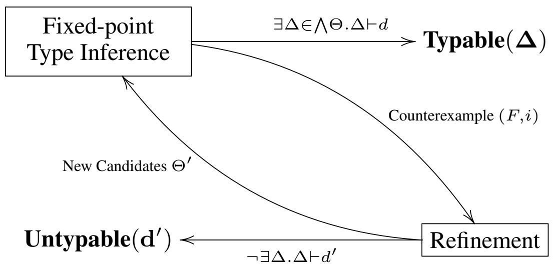
Fixed-point Type Inference
$\exists\Delta\in\bigwedge\Theta.\Delta\vdash d$
Typable($\boldsymbol{\Delta}$)
Counterexample (F,i)
New Candidates Θ'
Untypable($\mathbf{d}'$)
$\neg\exists\Delta.\Delta\vdash d'$
Refinement

Figure 10. The type inference CEGAR loop.

It is worth noting that the theorem would hold true even if Least had been used instead of Interp. Nonetheless, the idea is that Interp is likely to eliminate other spurious counterexamples that we would see in future had we used Least instead.

6. Putting Type Inference Components Together

Figure 10 summarizes our CEGAR-inspired type inference process. The fixed-point type inference algorithm checks if the program d is typable under the current set of candidates Θ', and if so, returns **Typable** with the inferred typing Δ, and otherwise, passes the counterexample (F, i) to the refinement algorithm. The refinement algorithm creates the unwinding d' from the counterexample and decides d''s typability completely. If d' is found untypable, then the refinement algorithm returns **Untypable** with the real counterexample d', and otherwise, returns new candidates Θ' that is sufficient for typing d'. The fixed-point routine refines the candidates by updating $\Theta := \Theta \uplus \Theta'$, and the CEGAR iteration repeats.

As remarked before, while the fixed point type inference and refinement algorithms are each guaranteed to terminate, the CEGAR loop may iterate forever, producing an ever more refined set of candidates. But, the type inference is sound and complete in the sense that it always returns the correct answer when it terminates.

6.1 Initializing Candidates

The remaining question is about priming the CEGAR loop, that is, how to pick the initial set of candidates. In principle, any Θ such that each $\Theta(F)$ is a finite non-empty set of well-formed types can be used as the starting set of candidates.

One approach to building a sensible initial Θ is to run the refinement process with artificial counterexamples (F, i) for each F for some i, and take the point-wise union of Θ' produced from each counterexample as the initial Θ. The implementation discussed in Section 8 takes this approach with $i = 1$.

Another possible approach is to heuristically create the initial candidates by scanning the program text, for example, by using expressions appearing in branch conditions as the formulas in the refinement base types. This is also the approach taken in Rondon et al. [34] for building the domain of possible types. Finally, we may allow the programmer to suggest additional candidates (e.g., as type annotations).

6.2 Example

Recall the summation program d from Example 2.4. Suppose that the current set of candidates is Θ shown below:

$$\begin{aligned}
\Theta(\mathtt{sum}) = \{&\mathtt{int} \to \mathtt{int} \to (\mathtt{int} \to \star) \to \star,\\
&\{u{:}\mathtt{int} \mid u \geq 100\} \to \{u{:}\mathtt{int} \mid u \geq 0\} \to\\
&\qquad(\{u{:}\mathtt{int} \mid u \geq 100\} \to \star) \to \star\}\\
\Theta(\mathtt{main}) = \{&() \to \star\}\\
\Theta(\mathtt{check}) = \{&\{u{:}\mathtt{int} \mid u \geq 100\} \to \star\}
\end{aligned}$$

Then, $\mathit{InferNext}^2(\lambda F. \bigwedge \Theta(F)) = \Delta$ with $\Delta(\mathtt{main}) = \bigwedge \emptyset$, and so the counterexample $(\mathtt{main}, 2)$ is reported. Hence, we unwind from `main` twice, and obtain d' shown earlier in Figure 9.

Next, linear intersection type shape inference infers $\hat{\Delta}$ such that $\hat{\Delta} \hat{\vdash}^1 d'$. From $\hat{\Delta}$ we obtain the template $\ddot{\Delta}$ such that

$$\begin{array}{l}
\ddot{\Delta}(\mathtt{sum1}) = \\
\quad x{:}\{u{:}\mathtt{int} \mid P_1\} \to y{:}\{u{:}\mathtt{int} \mid P_2\} \to \\
\qquad k{:}(z{:}\{u{:}\mathtt{int} \mid P_3\} \to \star) \wedge (z{:}\{u{:}\mathtt{int} \mid P_4\} \to \star) \to \star \\
\ddot{\Delta}(\mathtt{sum2}) = \\
\quad x{:}\{u{:}\mathtt{int} \mid Q_1\} \to y{:}\{u{:}\mathtt{int} \mid Q_2\} \to \\
\qquad\qquad\qquad k{:}(z{:}\{u{:}\mathtt{int} \mid Q_3\} \to \star) \to \star
\end{array}$$

where the predicate variables are fresh. (We omit the templates for `main1`, `check1`, and `check2`.) Generating constraints and solving for the least solutions, we obtain

$$\begin{array}{ll}
\mathit{Least}(P_1) \Leftrightarrow u = 100 & \mathit{Least}(Q_1) \Leftrightarrow u = 99 \\
\mathit{Least}(P_2) \Leftrightarrow u = 0 & \mathit{Least}(Q_2) \Leftrightarrow u = 100 \\
\multicolumn{2}{c}{\mathit{Least}(P_3) \Leftrightarrow \mathit{Least}(P_4) \Leftrightarrow \mathit{Least}(Q_3) \Leftrightarrow \bot}
\end{array}$$

As remarked in Section 5.3.1, these solutions, while correct, are stronger than desired. It is possible, though not guaranteed, for an interpolating theorem prover to generate *Interp* such that

$$\begin{array}{l}
\mathit{Interp}(Q_1) \Leftrightarrow \mathit{Interp}(Q_2) \Leftrightarrow \top \\
\mathit{Interp}(Q_3) \Leftrightarrow u \geq y
\end{array}$$

From $\mathit{Interp}(\ddot{\Delta})$, we obtain the new candidates Θ' such that

$$\begin{array}{l}
\Theta'(\mathtt{sum}) = x{:}\mathtt{int} \to y{:}\mathtt{int} \to \\
\qquad\qquad\qquad (\{u{:}\mathtt{int} \mid u \geq y\} \to \star) \to \star
\end{array}$$

Then, as shown in Example 2.4, there exists $\Delta \in \bigwedge(\Theta \uplus \Theta')$ such that $\Delta \vdash d$, and the fixed-point type inference algorithm returns $\mathbf{Typable}(\Delta)$.

7. Beyond Assertion Checking

We have shown that our system can be used to check the absence of assertion failures. This section presents an extension that checks more general program properties specified via user-provided types. Specifically, we allow the user to provide a mapping Υ from function names to types and ask if there exists a typing $\Delta \vdash d$ such that $\emptyset; \top \vdash \Delta(F) \leq \Upsilon(F)$ for all F. Such a specification conformance check can be handled by a simple extension outlined below. Note that this can also be used to force the system to infer a typing of a desired precision.

We assume that $\Upsilon(F)$ is well-formed for each F. We modify the fixed-point type inference algorithm from Section 4 so that *InferNext* reports the counterexample (F, i) if it reaches Δ such that $\emptyset; \top \not\vdash \Delta(F) \leq \Upsilon(F)$ at the ith fixed-point iteration. Otherwise, a fixed point Δ such that for all F, $\emptyset; \top \vdash \Delta(F) \leq \Upsilon(F)$ is reached and the system declares that d conforms to Υ.

We also extend the refinement algorithm from Section 5 so that given the counterexample (G, i), it unwinds d to get d' as before, and then checks if there exists Δ such that $\Delta \vdash d'$ and for all F, for all $F' \in \mathit{Inames}(F)$, $\emptyset; \top \vdash \Delta(F') \leq \Upsilon(F)$. If so, the refinement algorithm returns new candidates Θ' constructed from such Δ, and otherwise, declares that d does not conform to Υ. Note that the additional conditions $\emptyset; \top \vdash \Delta(F') \leq \Upsilon(F)$ can be reduced to constraints that can be solved by the constraint solving algorithm.

8. Implementation and Experiments

We have implemented a prototype of the type inference system, Depcegar, which takes a subset of Ocaml programs corresponding to the simple functional language (cf. Figure 2) extended to direct-style syntax. Depcegar handles non-CPS expressions by extending the type system with conditional types and union types, and handles non-A-normal forms by online A-normalization [13].

Depcegar is implemented as a modification to the Ocaml 3.10.2 compiler. We use Ocaml's parser and ML-type inference as the front-end to parse the program and obtain the simple types.[11] We use CSIsat 1.2 [5] as the interpolating theorem prover. CSIsat supports the quantifier-free first-order theory of uninterpreted function symbols and linear arithmetic (EUF+LA). CSIsat supports real arithmetic but not integer arithmetic, and so integers are approximated as reals in Depcegar (it does not affect the examples in this paper). For convenience, we use CSIsat both to generate interpolants in the refinement phase and to decide base-type subtyping judgements in the fixed-point type inference phase (i.e., **SubB** from Figure 7). The implementation contains about 5000 lines of original code. A web demo of Depcegar and the benchmark programs are available online [36].

We have conducted experiments on small hand-crafted programs, including the ones used as examples in the paper. Table 1 summarizes the results. Here, the first column is the program name and the second column is the running time in seconds. The column T-TP is the fraction of the running time spent by the interpolating theorem prover CSIsat (both interpolant computation and subtyping judgements), and the column T-INT is the fraction of the running time CSIsat spent computing interpolants. The times do not include the parsing and ML-type inference time. The column Candidates shows the total number of candidates generated (for all functions combined). The experiments were conducted on a Intel Core 2 Extreme 3GHz machine with 2GB of ram, running Linux.

Program	Time	T-TP	T-INT	Candidates
boolflip	0.2	95%	72%	10
sum	0.1	97%	26%	6
sum-acm	0.5	98%	14%	9
sum-all	0.1	93%	46%	9
mult	0.6	99%	31%	6
mult-cps	1.0	98%	31%	13
mult-all	0.5	98%	50%	10

Table 1. Experiment results - typable programs.

Program	Time	T-TP	T-INT	Candidates
boolflip-e	0.2	95%	79%	8
sum-e	2.8	99%	85%	9
sum-acm-e	9.9	99%	94%	10
sum-all-e	0.3	97%	75%	9
mult-e	13.9	99%	91%	9
mult-cps-e	42.1	99%	90%	16
mult-all-e	1.1	98%	71%	10

Table 2. Experiment results - untypable programs.

All of the programs in Table 1 are typable. We briefly describe each program. The program `boolflip` is the boolean program from Example 4.3. The program `sum-acm` is the summation program from Example 2.4, `sum` is the same summation program written in direct-style (i.e., without using the accumulation parameter y), and `sum-all` is `sum` recursively applied to all non-negative integers (i.e., it asserts $x \leq \sum_{i \in \{0,\ldots,x\}} i$ for all $x \geq 0$). The program `mult` is the multiplication program from Figure 1, `mult-cps` is the same program written in CPS, and `mult-all` is `mult` recursively

[11] Depcegar does not take advantage of ML's parametric polymorphism, but in principle, let polymorphism can be handled by inlining.

applied to all non-negative integers, that is, it replaces the last line of `mult` with the following:[12]

```
and f y = assert (y <= mult y y); f (y+1)
and main () = f 0
```

Depcegar was able to successfully infer a typing for each of the programs. The Candidates column shows that relatively few candidates are generated to type the programs, confirming our hypothesis that the interpolation-based candidate generation method is quite effective at generating good candidates.

To test Depcegar on untypable programs, we injected assertions errors into each of the programs. For example, `mult-e` replaces the last line of `mult` with the following to assert that $600 \leq 100 \times 5$:

```
and main () = assert (600 <= mult 100 5)
```

and `boolflip-e` applies F to $y\,y$ instead of $y\,x$ inside the body of F (cf. Example 4.3).

Depcegar successfully detected the type error in all of the programs. Table 2 summarizes the results. Note that the interpolation fraction T-INT tends to be higher for the untypable programs. This is primarily because their run ends during the last refinement phase, whereas for typable programs, the run ends when the last fixed-point type inference phase has finished. The results also show that Depcegar quickly detects that `boolflip-e` (and `sum-all-e` and `mult-all-e`) are untypable, but is quite slow on `mult-e` (and `sum-acm-e` and `mult-cps-e`).

More generally, we have observed that while Depcegar can often quickly detect typable programs to be typable by generating good candidates early in the CEGAR loop, for untypable programs, it must iterate the CEGAR loop long enough until a real counterexample is encountered. This may result in a large unwinding for a type error that only occurs "deep" in the program. A similar issue occurs in CEGAR-based model checking when detecting errors that take many steps from the initial state to reach [2, 25]. One possible remedy is to multiply the unwinding depth by an increasing factor as the CEGAR loop progresses. We leave for future work to address the issue in a more depth.

We also observe that Depcegar's running times are almost completely dominated by that of theorem proving, with a non-negligible fraction dedicated to computing interpolants. As discussed in Section 9.3, interpolating theorem provers are fairly new technology and are actively being researched. Hence, we expect the running times to improve as interpolating theorem provers mature. A possible optimization is to use a faster, non-interpolating theorem prover for deciding subtyping judgements and use an interpolating theorem prover only for computing interpolants.

Finally, we note that while the interpolation-based refinement guarantees the elimination of the given spurious counterexample, it does not guarantee the convergence of our system on all typable programs.[13] An interesting future research direction is to investigate a more complete approach to candidate generation.

9. Related Work

9.1 Inferring Dependent Types

Inferring complex types via fixed-point type inference iteration is a classic idea. For instance, Freeman and Pfenning [14] infers the strongest refinement types given a user-provided lattice of refinement types. The recent work by Rondon et al. [34] can be casted as an instance of this approach to dependent types. Their system chooses a finite set of candidate formulas by a syntactic scan of the program text and the user-provided set of formulas, and infers the strongest types within the lattice of dependent types confined to these formulas via fixed-point iteration and theorem proving, similar to the fixed-point type inference phase of our system. Lacking automatic refinement, these approaches require the lattice of candidate types to be pre-defined and be of finite height. In contrast, our system automatically infers candidate types within an infinite domain of types (i.e., unbounded intersections and arbitrary formulas) via counterexample analysis.

One advantage of our approach is that the type inference becomes complete. That is, when the system declares the program to be untypable, the user is assured that it is actually untypable rather than wondering if more candidates were needed. A consequence of this is that the inferred types may not be the strongest. But, this is generally unavoidable as the strongest types may not even be finitely expressible within the underlying theory, as in, for example, the `mult` program from Figure 1. However, as remarked in Section 7, the system can be made to infer types of the user-specified strength.

Concurrent to our work, Unno and Kobayashi [37] have proposed to infer dependent types via interpolation and iterative unrolling of recursive constraints. Chin et al. [6, 7] have also suggested a constraint unrolling approach with the Omega test [33] as the backend solver. These approaches use neither candidate types nor a fixed-point type inference routine, but they resemble the refinement phase of our work in that they also reduce the inference problem to finding a solution to a set of first-order logic constraints. One issue with these purely constraint-based approaches is the presence of "false constraint cycles" like the one shown in Example 5.6.[14] In contrast, our approach divides type inference into the fixed-point type inference phase and the constraint-solving refinement phase so that the latter is able to leverage unwinding and linear intersection types to ensure constraint acyclicity.

We note that none of the previous systems listed above supports unbounded intersection types. To the best of our knowledge, our system is the first dependent (or refinement) type inference system that can infer unbounded intersection types embedded with arbitrary formulas from a first-order theory.

9.2 Inferring Intersection Types

The success of our system owes much to intersection types. Not only do intersection types make the underlying dependent type system more expressive by being able to type more safe programs, they are also crucial to the refinement phase of the system that uses linear intersection type inference [21] to infer a derivation shape that is sufficient for ensuring both type inference completeness and constraint acyclicity.

Linear intersection type inference cannot be applied directly to programs containing recursive definitions as linear intersection types are not even defined for such programs. Our approach circumvents the issue by iteratively producing non-recursive program fragments as counterexamples, and then checking if the candidate types inferred for the fragments are also sufficient for typing the original, recursive program.

In our system, intersection types are restricted to only intersect types of the same simple-type shape, but intersection type inference algorithms are capable of inferring arbitrary intersection of types, as well as inferring principal typing [38], which we also do not utilize in this work. We leave for future work to capitalize on the full potential of intersection type inference.

[12] Depcegar does not handle arithmetic overflows.

[13] However, it is trivial to show that the system is guaranteed to converge on programs with only finitary-data base types (e.g., just booleans).

[14] For [6, 7], if extended to higher-order functions.

9.3 Model Checking

Counterexample-guided abstraction refinement has been used with great success in hardware and software model checking (see, e.g., [3, 9, 16]). However, most of the existing software model checkers only target low-level imperative programs such as device drivers written in C, and are unsuitable for functional programs because they cannot accurately model higher-order functions and function closures.

Recently, researchers have proposed model checking algorithms for typed higher-order functional programs by leveraging their equivalence to higher-order pushdown systems [24, 31]. However, these algorithms only handle finite data domains, whereas our system supports infinite data domains such as integers by utilizing an interpolating theorem prover.

Interpolation has found various applications in model checking, such as predicate abstraction [15, 18] and reachable state approximation [26, 28]. We have shown that interpolation is also quite effective for inferring dependent types. Algorithms for computing interpolants for various theories are actively being researched (e.g., [5, 8, 17, 20]). As future work, we plan to extend our system to other data types, such as lists and arrays, by using interpolating theorem provers for their theories.

10. Conclusion

We have presented a new approach to inferring dependent types. The key to the success is the iterative refinement of candidate dependent types via counterexample analysis, utilizing linear intersection type inference and interpolation. We have shown that the approach enables a sound and complete type inference of an expressive dependent type system that allows unbounded intersection types embedding arbitrary formulas from a first order theory.

References

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

[2] T. Ball, O. Kupferman, and M. Sagiv. Leaping loops in the presence of abstraction. In *CAV*, pages 491–503, 2007.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, pages 17–32, 2008.

[5] D. Beyer, D. Zufferey, and R. Majumdar. CSIsat: Interpolation for LA+EUF. In *CAV*, pages 304–308, 2008.

[6] W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[7] W.-N. Chin, S.-C. Khoo, and D. N. Xu. Extending sized type with collection analysis. In *PEPM*, pages 75–84, 2003.

[8] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theories. In *TACAS*, pages 397–412, 2008.

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

[10] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, pages 520–535, 2007.

[11] W. Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[12] C. Flanagan. Hybrid type checking. In *POPL*, pages 245–256, 2006.

[13] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247, 1993.

[14] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, pages 268–277, 1991.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[17] H. Jain, E. M. Clarke, and O. Grumberg. Efficient Craig interpolation for linear Diophantine (dis)equations and linear modular equations. In *CAV*, pages 254–267, 2008.

[18] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV*, pages 39–51, 2005.

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

[20] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *SIGSOFT FSE*, pages 105–116, 2006.

[21] A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theor. Comput. Sci.*, 311(1-3):1–70, 2004.

[22] K. Knowles and C. Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *PLPV*, pages 27–38, 2009.

[23] K. W. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, pages 505–519, 2007.

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

[25] D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *CAV*, pages 152–165, 2006.

[26] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.

[27] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[28] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[29] P. M. Neergaard. Theoretical pearls: A bargain for intersection types: a simple strong normalization proof. *J. Funct. Program.*, 15(5):669–677, 2005.

[30] P. M. Neergaard and H. G. Mairson. Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In *ICFP*, pages 138–149, 2004.

[31] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90, 2006.

[32] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[33] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, pages 4–13, 1991.

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

[35] R. Statman. The typed lambda-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9:73–81, 1979.

[36] T. Terauchi. Dependent types from counterexamples, 2009. http://www.kb.ecei.tohoku.ac.jp/∼terauchi/.

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

[38] J. B. Wells. The essence of principal typings. In *ICALP*, pages 913–925, 2002.

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

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

A. Simple Type System

We formally present the simple type system. The syntax of simple types is defined by the following grammar: (Also shown in Sec-

$$\frac{}{\Gamma \vdash^s F : \Gamma(F)} \quad \frac{}{\Gamma \vdash^s c : \mathit{simple}(\mathit{ty}(c))}$$

$$\frac{}{\Gamma \vdash^s x : \Gamma(x)} \quad \frac{\Gamma \vdash^s e : \texttt{bool}}{\Gamma \vdash^s \texttt{assert}\, e : \texttt{int}}$$

$$\frac{\Gamma \vdash^s e_1 : s \quad \Gamma, x{:}s \vdash^s e_2 : \star}{\Gamma \vdash^s \texttt{let}\, x = e_1\, \texttt{in}\, e_2 : \star} \quad \frac{\Gamma \vdash^s e : s \to s' \quad \Gamma \vdash^s y : s}{\Gamma \vdash^s e\, y : s'}$$

$$\frac{\Gamma \vdash^s x : \texttt{bool} \quad \Gamma \vdash^s e_1 : \star \quad \Gamma \vdash^s e_2 : \star}{\Gamma \vdash^s \texttt{if}\, x\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \star}$$

Figure 11. The simple type system type checking rules.

tion 2.1.)

$$\begin{array}{rcl} B & ::= & \texttt{int} \mid \texttt{bool} \\ s & ::= & \star \mid B \mid s \to s' \end{array}$$

Figure 11 shows the type checking rules. Here, we overload the symbols Γ and Δ to range over mappings to simple types.

DEFINITION A.1 (Simply-typed program). *Let Δ be a top-level simple type environment. We write $\Delta \vdash^s d$ if for each function $F\,\overrightarrow{x} = e \in d$, we have $\Delta, \overrightarrow{x{:}s} \vdash^s e : \star$ where $\Delta(F) = \overrightarrow{x{:}s} \to \star$.*

We say that a program d is simply-typed if there exists Δ such that $\Delta \vdash^s d$.

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

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 [39]. (Recall that due to CPS, a program does not return.)

THEOREM A.2 (Simple Type Safety). *Let d be simply-typed and $\texttt{main}\,() = e_{\texttt{main}} \in d$. Then, for any e such that $e_{\texttt{main}} \to_d^* e$, either $e \to_d \mathit{fail}$ or there exists e' such that $e \to_d e'$.*

B. Sound Constant Types

For a constant operation c over booleans and/or integers, we say that the dependent type $\tau = \overrightarrow{x{:}B} \to \{u{:}B \mid \phi\}$ is *sound* if the following properties all hold:

- τ is closed.
- $\mathit{simple}(\tau) = \mathit{sty}(c)$.
- For any $\overrightarrow{v}$ in the input domain of c, $\phi[\overrightarrow{v}/\overrightarrow{x}][[\![c]\!](\overrightarrow{v})/u]$ is logically valid.

Sound types for boolean and integer constants are defined analogously. For example, a sound type for the constant c of the base type B is $\{u{:}B \mid u = c\}$.

While a constant operator in our language accepts all inputs in its domain, behaviors such as the divide-by-0 error can be modeled by inserting appropriate assert statements.

C. Proofs of Key Results

We prove Theorem 5.4, Theorem 5.5, and Theorem 5.8. To facilitate the proofs of Theorem 5.4 and Theorem 5.5, we define a β reduction like semantics for unwound program fragments.

$$C[(\texttt{st}\,\ell\,\texttt{in}\,\lambda p.e)\,q] \to_\beta \\ \quad C[\texttt{st}\,\ell :: (b = q)\,\texttt{in}\,e[b/p]]$$

$$C_1[\texttt{let}\,a = e\,\texttt{in}\,C_2[(\texttt{st}\,\ell\,\texttt{in}\,\lambda p.e')\,a]] \to_\beta \\ \quad C_1[\texttt{let}\,a = e\,\texttt{in}\,C_2[\texttt{st}\,\ell :: (b = e)\,\texttt{in}\,e'[b/p]]]$$

$$C_1[\texttt{st}\,\ell_1 :: a = e :: \ell_2\,\texttt{in}\,C_2[(\texttt{st}\,\ell\,\texttt{in}\,\lambda p.e')\,a]] \to_\beta \\ \quad C_1[\texttt{st}\,\ell_1 :: a = e :: \ell_2\,\texttt{in}\,C_2[\texttt{st}\,\ell :: (b = e)\,\texttt{in}\,e'[b/p]]]$$

$$C_1[\texttt{let}\,a = e\,\texttt{in}\,C_2[a\,x]] \to_\beta C_1[\texttt{let}\,a = e\,\texttt{in}\,C_2[e\,x]]$$

$$C_1[\texttt{st}\,\ell_1 :: a = e :: \ell_2\,\texttt{in}\,C_2[a\,x]] \\ \quad \to_\beta C_1[\texttt{st}\,\ell_1 :: a = e :: \ell_2\,\texttt{in}\,C_2[e\,x]]$$

Figure 12. The $\to_\beta$ reduction rules.

First, we update the syntax of the language so that λ abstractions are used to denote functions and function closures:

$$\begin{array}{rcl} e & ::= & x \mid c \mid F \mid \lambda p.e \mid e\,x \mid \texttt{let}\,a = e_1\,\texttt{in}\,e_2 \\ & \mid & \texttt{if}\,x\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \mid \texttt{assert}\,e \mid \texttt{st}\,\ell\,\texttt{in}\,e_2 \\ x & ::= & p \mid a \\ \ell & ::= & \overrightarrow{a = e} \end{array}$$

(For convenience, we overload the meta variable e to range over the expressions in the updated syntax.) We separate variables to two kinds: *stack variables*, ranged over by symbols a, b, etc., and *parameter variables*, ranged over by symbols p, q, etc. Symbols x, y, etc. range over both kinds. Stack variables are bound at `let` and the new construct `st`. Conceptually, the expression $\texttt{st}\,\overrightarrow{a = e}\,\texttt{in}\,e$ represents a function-call stack frame with the stack $\overrightarrow{a = e}$, and the program counter e. We use stacks instead of explicit substitution so that we can type intermediate states via the **App** and **If** rules as they require a variable for the argument and the branch condition. Stack frames are also used to mark function bodies to avoid base-type variables in the parent context from affecting their typing.

We translate an unwound fragment d' to its representation in the updated syntax $\mathit{lam}(d')$ by replacing all non-leaf functions with the equivalent λ representation enclosed in an empty stack frame, and replacing all let-bound variables with a fresh stack variable. More formally, for $F\,\overrightarrow{x} = e \in d'$, we define $\mathit{lam}(F) = \texttt{st}\ \varepsilon\ \texttt{in}\ \lambda\overrightarrow{p}.\mathit{lam}(e[\overrightarrow{p}/\overrightarrow{x}])$ where $\overrightarrow{p}$ are fresh, and $\mathit{lam}(e)$ is defined inductively as e with all its non-leaf free function name G replaced by $\mathit{lam}(G)$ and all let bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ replaced by $\texttt{let}\ a = e_1\ \texttt{in}\ e_2[a/x]$ for a fresh stack variable a. (Here, ε denotes the empty sequence, and $\lambda\overrightarrow{x}.e$ is a short hand for $\lambda x_1.\lambda x_2.\ldots\lambda x_n.e$ where $\overrightarrow{x} = x_1, x_2, \ldots, x_n$.) Then, we define $\mathit{lam}(d') = \mathit{lam}(F)$ where F is the root function of d'. Note that only the leaf functions of d' may appear free in $\mathit{lam}(d')$.

We define *normal forms*, ranged over by *nf*, by the grammar below:

$$\begin{array}{rcl} \mathit{nf} & ::= & x \mid p\,\overrightarrow{x} \mid c\,\overrightarrow{x} \mid F\,\overrightarrow{x} \mid \lambda p.\mathit{nf} \mid \texttt{st}\,\mathit{n\ell}\,\texttt{in}\,\mathit{nf} \\ & \mid & \texttt{let}\,a = \mathit{nf}_1\,\texttt{in}\,\mathit{nf}_2 \mid \texttt{assert}\,\mathit{nf} \\ & \mid & \texttt{if}\,x\,\texttt{then}\,\mathit{nf}_1\,\texttt{else}\,\mathit{nf}_2 \\ \mathit{n\ell} & ::= & \overrightarrow{a = \mathit{nf}} \end{array}$$

Let C range over any expression context. Figure 12 defines the "β reduction" semantics for expressions in the updated syntax. Intuitively, these rules implement the usual β reduction, but using stacks instead of explicit substitutions. In the first three rules, the stack variable b appearing to the right of $\to_\beta$ is fresh (i.e., it does not appear free to the left). The first three rules, differing only on whether the variable a (or q) is a parameter variable, let-bound or st-bound, applies the function $\texttt{st}\ \ell\ \texttt{in}\ \lambda p.e$ to the argument a (or

$$\frac{\Gamma, p{:}\sigma; \theta \vdash e : \tau}{\Gamma; \theta \vdash \lambda p.e : p{:}\sigma \to \tau}\ \textbf{Lam}$$

$$\frac{\begin{array}{c}\text{For each } i.\Gamma; \theta \vdash_\wedge e_i : \sigma_i \\ a_1{:}\sigma_1, \ldots, a_n{:}\sigma_n; \top \vdash e' : \star\end{array}}{\Gamma; \theta \vdash \texttt{st}\, a_1 = e_1, \ldots, a_n = e_n \,\texttt{in}\, e' : \star}\ \textbf{St}$$

$$\frac{\begin{array}{c}\Gamma, p{:}\sigma \setminus X; \theta \vdash^1 e : \tau \\ X = \{p \mid p \notin \mathit{free}(e) \text{ and } \mathit{sty}(p) \text{ not base}\}\end{array}}{\Gamma; \theta \vdash^1 \lambda p.e : p{:}\sigma \to \tau}\ \textbf{Lam}^\mathbf{1}$$

$$\frac{\begin{array}{c}\text{For each } i.\Gamma; \theta \vdash^1_\wedge e_i : \sigma_i \\ a_1{:}\sigma_1, \ldots, a_n{:}\sigma_n \setminus X; \top \vdash^1 e' : \star \\ X = \{a_i \mid a_i \notin \mathit{free}(e) \text{ and } \mathit{sty}(a_i) \text{ not base}\}\end{array}}{\Gamma; \theta \vdash^1 \texttt{st}\, a_1 = e_1, \ldots, a_n = e_n \,\texttt{in}\, e' : \star}\ \textbf{St}^\mathbf{1}$$

Figure 13. Dependent type checking rules for the extended expressions.

q). The last two rules, differing only on whether the variable a is let-bound or st-bound, substitutes a with e.

Note that the semantics do not reduce conditionals, asserts, constant applications, nor leaf function applications, but reduces user-defined function applications everywhere. It is easy to see that a normal form cannot be reduced, i.e.,

LEMMA C.1. *For any normal form nf, there exists no e such that $nf \to_\beta e$.*

We update the simple type system $\vdash^s$ in a straightforward way to the extended syntax by adding the rules below.

$$\frac{\Gamma, p{:}s \vdash^s e : s'}{\Gamma \vdash^s \lambda p.e : s \to s'}$$

$$\frac{\begin{array}{c}\text{For each } i.\Gamma \vdash^s e_i : s_i \\ a_1{:}s_1, \ldots, a_n{:}s_n \vdash^s e' : \star\end{array}}{\Gamma \vdash \texttt{st}\, a_1 = e_1, \ldots, a_n = e_n \,\texttt{in}\, e' : \star}\ \textbf{St}$$

The next lemma states the expected, that is, simple typability of d' and $lam(d')$ coincides.

LEMMA C.2. *Let d' be an unwound program fragment. Then, $\exists\Delta.\Delta \vdash^s d'$ if and only if $\exists\Delta.\Delta \vdash^s lam(d')$*

Proof: By induction on the structure of d'. □

It can be shown that $\vdash^s$ typing is preserved across $\to_\beta$ using the standard method [39].

LEMMA C.3. *If $\Gamma \vdash^s e : \star$ and $e \to_\beta e'$, then $\Gamma \vdash^s e' : \star$.*

Also, by the standard strong normalization result for the simply typed λ calculus, we have that $\to_\beta$ is strongly normalizing for simply typable terms. (See, e.g., [29].)

LEMMA C.4. *Suppose $\Gamma \vdash^s e : \star$. Then, e strongly normalizes to some normal form.*

Henceforth, we assume that any expression is simply typed, and that its simple type, $\mathit{sty}(e)$, is available.

We also update the dependent type system $\vdash$ and $\vdash^1$ to the extended expressions so that the typability of d' and $lam(d')$ also coincides there. Formally, we add the rules shown in Figure 13. Note that the typing rules for a stack frame ($\mathbf{St}$ and $\mathbf{St^1}$) ignores the parent's environment assumption.

We restate the expected property as a lemma.

LEMMA C.5. *Let d' be an unwound program fragment. Then, $\exists\Delta.\Delta \vdash d'$ if and only if $\exists\Delta.\Delta; \top \vdash lam(d')$, and $\exists\Delta.\Delta \vdash^1 d'$ if and only if $\exists\Delta.\Delta; \top \vdash lam(d')$.*

Proof: By induction on the structure of d'. □

It is easy to show the preservation result for the dependent type system $\vdash$, again via the standard method [39].

LEMMA C.6. *If $\Gamma \vdash e : \star$ and $e \to_\beta e'$, then $\Gamma \vdash e' : \star$.*

Now, let us cast the extended $\vdash^1$ type rules as constraint generation rules by modifying the $\mathbf{SubB}$ rule just as we have done for the original set of rules in Section 5.3. We show that the generate set of constraints coincides for d' and $lam(d')$. For convenience, we assume that the derivation is modified so that the constraint generated by each subderivation is written explicitly as in $\Gamma; \phi \vdash^1 e : \tau; \mathcal{C}$ where $\mathcal{C}$ is the set of constraints generated by the derivation $\Gamma; \phi \vdash^1 e : \tau$. To generate constraints from $lam(d')$, it suffices to just make fresh predicate variables appear everywhere in the argument type σ at $\mathbf{Lam^1}$.

The following theorem states that constraints generated from d' is equivalent to those generated from $lam(d')$.

LEMMA C.7. *Let $\mathcal{C}$ be the set of constraints generated from $\Delta \vdash^1 d'$. Then, we have $\Delta \setminus \{F \mid F \text{ is not leaf}\}; \top \vdash^1 lam(d') : \star; \mathcal{C}$.*

Proof: By induction on the structure of d'. □

C.1 Theorem 5.4

THEOREM 5.4. *Let d' be an unwound program fragment. Then, the following are equivalent.*

(1) There exists Δ such that $\Delta \vdash^1 d'$.
(2) There exists Δ such that $\Delta \vdash d'$.

By Lemma C.5, it suffices to show that the typability of $lam(d')$ coincides for $\vdash^1$ and $\vdash$. (1) $\Rightarrow$ (2) is trivial because any $\vdash^1$ typing derivation is a valid $\vdash$ derivation that just does not use function type bindings non-linearly.

We prove (2) $\Rightarrow$ (1) by showing that any strongly-normalizing $\vdash$ typable terms are $\vdash^1$ typable. Then, because any $lam(d')$ is strongly normalizing by Lemma C.4 anyway, it follows that any $\vdash$ typable term is $\vdash^1$ typable.

For a free non-base type stack variable a in nf, we write $I(a, nf)$ for the set of types σ such that $\mathit{simple}(\sigma) = \mathit{sty}(a)$ and the top level linear intersections of σ matches the occurrences of a in nf. For example,

$$I(a, \lambda p.p\, a\, a) = \{\tau \wedge \tau' \mid \mathit{simple}(\tau) = \mathit{simple}(\tau') = \mathit{sty}(a)\}$$

For a type environment Γ, let $I(\Gamma, nf)$ be the set of type environments Γ' that maps all base-type variables and (non-base-type) stack variables in $\mathit{dom}(\Gamma)$ (i.e., it does not map non-base-type parameter variables and function names) such that $\Gamma'(a) \in I(a, nf)$ for $a \in \mathit{dom}(\Gamma)$ a non-base-type stack variable and $\Gamma'(x) = \langle\!|\Gamma(x)|\!\rangle$ for $x \in \mathit{dom}(\Gamma)$ a base type variable, where $\langle\!|\bigwedge_i\{u{:}B \mid \theta_i\}|\!\rangle = \{u{:}B \mid \bigwedge_i \theta_i\}$. Essentially, $\Gamma' \in I(\Gamma, nf)$ is a type environment that is equivalent to Γ for base-type variables and has arbitrary well-shaped types for non-base-type stack variables.

We now show that any $\vdash$ typable normal form is $\vdash^1$ typable under such an environment.

LEMMA C.8. *Suppose $\Gamma; \top \vdash_\wedge nf : \sigma$. Let $\Gamma' \in I(\Gamma, nf)$. Then, there exists $\Gamma'' \supseteq \Gamma'$ and σ' such that $\Gamma''; \top \vdash^1_\wedge nf : \sigma'$, $\mathit{simple}(\sigma') = \mathit{simple}(\sigma)$, and if σ is a base type, $\sigma' = \langle\!|\sigma|\!\rangle$.*

Proof: By induction on the structure of nf. The case nf is a variable is trivial. For the case nf is $p\,\overrightarrow{x}$ (resp. $F\,\overrightarrow{x}$), we set $\Gamma''(p)$ (resp.

$\Gamma''(F)$) appropriately so that it is a function taking the arguments of the type $\Gamma'(\overrightarrow{x})$ (if an argument is also a function-type parameter variable or a function name, then it too can be set to be of any appropriate type).

The case nf is $c\,\overrightarrow{x}$ follows from the sound constant type assumption (cf. Section B) that all arguments to a constant operator are of base types. The case nf is a lambda abstraction follows trivially from the induction hypothesis.

For the case nf is a let expression $\mathtt{let}\ a = nf_1\ \mathtt{in}\ nf_2$, we split on whether $sty(a)$ is a function type or not, and apply induction hypothesis. The case nf is a stack frame $\mathtt{st}\,n\ell\,\mathtt{in}\,nf'$ is similar. The case nf is an assert expression or a conditional expression follows from induction hypothesis and the fact that asserted expressions and branch conditions are always of base types. □

Next, we show the "subject-expansion" property.

LEMMA C.9. *Suppose* $e_1 \rightarrow_\beta e_2$*,* $\Gamma_0; \phi \vdash e_1 : \star$*, and* $\Gamma; \phi \vdash^1 e_2 : \star$*. Then, there exists* Γ' *such that* $\Gamma'; \phi \vdash^1 e_1 : \star$*.*

Proof: We prove by case analysis on the reduction kinds. For the case the reduction is one of the first three rules in Figure 12, we type e_1 by setting the type of the parameter variable p to be the type of b in the typing for e_2.

The case the reduction is one of the last two rules in Figure 12 follows trivially from the inspection of the rules $\mathbf{St^1}$ and $\mathbf{Let^1}$. □

By Lemmas C.4,C.6,C.8, and C.9, it follows that if $\Delta; \top \vdash e : \star$ then there exists Δ' such that $\Delta'; \top \vdash^1 e : \star$. This proves (2) $\Rightarrow$ (1).

C.2 Theorem 5.5

THEOREM 5.5. *The generated set of constraints* $\mathcal{C}$ *does not contain constraints of the form* $\{P_i\rho_i \wedge \ddot{\theta}_i \Rightarrow P_{i+1}\rho'_i \mid 1 \leq i \leq n\}$ *for some* $n \geq 1$ *with* $P_1 = P_{n+1}$*.*

By Lemma C.7, it suffices to show that $lam(d')$ generates acyclic constraints. Because we are only concerned with constraint-generation in this section, we assume that $\vdash^1$ judgements and derivations are that of constraint generation and not type checking. Recall that constraint generation judgements have the form $\Gamma; \phi \vdash^1 e : \tau; \mathcal{C}$ where $\mathcal{C}$ is the set of constraints generated by the derivation (cf. Section C). Also, for readability, we elide the decorated notation $\ddot{\theta}$, $\ddot{\tau}$, $\ddot{\Delta}$, etc., which was used in the main body of the paper to emphasize the fact that the object may contain predicate variables, and simply use the non-decorated version θ, τ, Δ, etc.

We prove the theorem semantically, like (2) $\Rightarrow$ (1) of Theorem 5.4. This time, we show that any strongly normalizing (w.r.t. $\rightarrow_\beta$) expression generates acyclic constraints. Then, the result follows from Lemma C.4 which says that any unwound program fragment is actually strongly normalizing.

In a spirit similar to the construction of $I(\Gamma, nf)$ from Section C.1, for a normal form nf, let $I(nf)$ be the set of type environments Γ mapping base-type variables (both parameter and stack) and non-base-type stack variables free in nf to arbitrary types of the correct linear shape. For example,

$$\begin{aligned} &I(\lambda p.p\,a\,a) \\ &\quad = \{\{a \mapsto \tau \wedge \tau'\} \mid simple(\tau) = simple(\tau') = sty(a)\}\end{aligned}$$

We now show that the set of constraints generated from a normal form expression under such an environment is acyclic.

LEMMA C.10. *Let* $\Gamma \in I(nf)$ *and* $\mathcal{C}$ *be any acyclic set of constraints. Then, there exists* $\Gamma' \supseteq \Gamma$ *and* τ' *such that* $\Gamma'; \phi \vdash^1 nf : \tau'; \mathcal{C}'$ *and* $\mathcal{C} \cup \mathcal{C}'$ *is acyclic.*

Proof: By induction on the structure of nf. The case nf is a variable is trivial. The cases nf is an application $p\,\overrightarrow{x}$ (resp. $F\,\overrightarrow{x}$) holds by letting the type of p (resp. F) in Γ' to be of a type containing fresh predicate variables everywhere (i.e., distinct, and not in Γ or $\mathcal{C}$). Note that $p \notin dom(\Gamma)$ because p is a function-type variable. The constant application case $c\,\overrightarrow{x}$ is trivial.

The case nf is a lambda abstraction follows trivially from induction hypothesis. The case nf is a stack frame or a let expression is also straightforward from induction hypothesis, instantiating $\mathcal{C}$ to be the constraints from the other subterms. The case nf is an assert expression or a conditional expression is similar. □

Note that the lemma implies that constraints $\mathcal{C}$ generated from a program-level environment (i.e., $\Delta; \top \vdash^1 nf : \star; \mathcal{C}$ for Δ with fresh predicate variables everywhere.) is acyclic.

Given constraints $\mathcal{C}$ we say that there is an *edge* from P to Q if there exists a constraint of the form $P\rho \wedge \theta \Rightarrow Q\rho' \in \mathcal{C}$. We say that there is a *path* from P to Q if there exists a non-empty sequence of edges connecting P to Q. Note that $\mathcal{C}$ is acyclic if and only if there exists no path connecting a predicate variable to itself.

We now show the "subject-expansion" property.

LEMMA C.11. *Suppose* $e_1 \rightarrow_\beta e_2$*, and* $\Gamma; \phi \vdash^1 e_2 : \star; \mathcal{C}$ *with* $\mathcal{C}$ *acyclic. Then, there exists* Γ' *such that* $\Gamma'; \phi \vdash^1 e_1 : \star; \mathcal{C}'$ *with* $\mathcal{C}'$ *acyclic.*

Proof: We prove by case analysis on the reduction kinds. For the first three rules in Figure 12, we use fresh predicate variables for the type of the argument p.

Then, it can be shown that C' comprises of the predicate variables in C and the fresh predicate variables from the type of p. The new constraints from the function application in e_1 do not induce new paths in $\mathcal{C}'$ between variables that are in $\mathcal{C}$, and that there are no edges directly connecting the new predicate variables appearing in the type of p. Thus, $\mathcal{C}'$ is acyclic.

The case the reduction is one of the last two rules follows trivially from the inspection of the rules $\mathbf{St^1}$ and $\mathbf{Let^1}$. □

Lemma C.10 and Lemma C.11 imply that constraints generated from any strongly normalizing e is acyclic. Therefore, by Lemma C.7 and Lemma C.4, we have proven Theorem 5.5.

C.3 Theorem 5.8

THEOREM 5.8. *The algorithm computes Interp such that Interp* $\models \mathcal{C}$ *if and only if* $\mathcal{C}$ *is satisfiable.*

The only if direction is trivial. We prove the if direction. We enumerate the predicate variables as $P_1, \ldots, P_n$ where $P_i < P_{i+1}$ for $1 \leq i < n$. (Recall that predicate variables are totally ordered.)

Let S be a mapping from predicate variables to formulas. For $j, k \leq n$, we define S_j^k to be the following mapping:

$$S_j^k(P_i) = \begin{cases} Least(P_i) & \text{if } i < j \\ S(P_i) & \text{if } k < i \end{cases}$$

($S_j^k(P_i)$ is undefined for $j \leq i \leq k$.) Recall that a predicate variable can occur at most once in any constraint $\ddot{\theta} \Rightarrow \ddot{\theta}' \in \mathcal{C}$. For each P_i, let $\Phi_i \subseteq \mathcal{C}$ be the set of constraints of the form $P_i\rho \wedge \ddot{\theta} \Rightarrow \ddot{\theta}'$.

Let us write $Ok_i(S)$ if for all $j \geq i$, we have $Least(P_j) \Rightarrow S(P_j)$ and $S(P_j) \Rightarrow (S_j^j(\ddot{\theta}) \Rightarrow S(\ddot{\theta}'))\rho^{-1}$ for all $P_j\rho \wedge \ddot{\theta} \Rightarrow \ddot{\theta}' \in \Phi_j$. Note that S such that $Ok_1(S)$ is exactly *Interp*.

The following lemma shows that if $\mathcal{C}$ is satisfiable, then the algorithm is able to compute some *Interp*.

LEMMA C.12. *Suppose Least* $\models \mathcal{C}$*. Then, there exists* S *such that* $Ok_1(S)$*.*

Proof: We prove $Ok_i(S)$ for all i by induction in descending order.

$\boxed{\text{Base case}}$

We show that there exists S such that $Ok_n(S)$. Pick $P_n\rho \wedge \ddot{\theta} \Rightarrow \ddot{\theta}' \in \Phi_n$. By the property of the ordering $<$, it must be the case that $\ddot{\theta}'$ is concrete. Then, because $\mathit{Least} \models \mathcal{C}$, we have $\mathit{Least}(P_n)\rho \Rightarrow (\mathit{Least}(\ddot{\theta}) \Rightarrow \ddot{\theta}')\rho^{-1}$. So, there exists an interpolant $S(P_n) = \langle \mathit{Least}(P_n), (\mathit{Least}(\ddot{\theta}) \Rightarrow \ddot{\theta}')\rho^{-1}\rangle$ and we have $Ok_n(S)$.

$\boxed{\text{Inductive case}}$

Let S be such that $Ok_j(S)$ for all $j > i$ (by definition the same S can be used for all such j). We show that there exists S' such that $Ok_i(S')$.

Pick $P_i\rho \wedge \ddot{\theta} \Rightarrow \ddot{\theta}' \in \Phi_i$. Let $k > i$ be the smallest k such that P_k appears in $\ddot{\theta}$. The following is a tautology.

$$\mathit{Least}(P_i)\rho \Rightarrow (S_i^k(\ddot{\theta}) \wedge (S_{i+1}^k(P_i\rho \wedge \ddot{\theta}) \Rightarrow S(\ddot{\theta}')) \Rightarrow S(\ddot{\theta}'))$$

Let $P_k\rho_k \wedge \ddot{\theta}_k = \ddot{\theta}$. By induction hypothesis, we have $S(P_k)\rho_k \Rightarrow (S_{i+1}^k(P_i\rho \wedge \ddot{\theta}_k) \Rightarrow S(\ddot{\theta}'))$. Therefore, we have

$$\mathit{Least}(P_i)\rho \Rightarrow (S_i^k(\ddot{\theta}) \Rightarrow S(\ddot{\theta}'))$$

Let $S'(P_i) = \langle \mathit{Least}(P_i), (S_i^k(\ddot{\theta}) \Rightarrow S(\ddot{\theta}')\rho^{-1}\rangle$ and $S'(P_j) = S(P_j)$ for all $j > i$. Then, $Ok_i(S')$.

Now suppose no such $k > i$ exists. Because $\mathit{Least} \models \mathcal{C}$, we have $\mathit{Least}(P_i)\rho \Rightarrow (\mathit{Least}(\ddot{\theta}) \Rightarrow \mathit{Least}(\ddot{\theta}'))$. By induction hypothesis, we have $\mathit{Least}(\ddot{\theta}') \Rightarrow S(\ddot{\theta}')$. Therefore,

$$\mathit{Least}(P_i)\rho \Rightarrow (\mathit{Least}(\ddot{\theta}) \Rightarrow S(\ddot{\theta}'))$$

And we can set $S'(P_i) = \langle \mathit{Least}(P_i), (\mathit{Least}(\ddot{\theta}) \Rightarrow S(\ddot{\theta}'))\rho^{-1}\rangle$ and $S'(P_j) = S(P_j)$ for all $j > i$, and get $Ok_i(S')$. □

To complete the proof of the theorem, we show that *Interp* obtained this way (i.e., $Ok_1(\mathit{Interp})$) is actually a solution for $\mathcal{C}$.

LEMMA C.13. *Suppose $Ok_1(S)$. Then, for all i, $S \models \Psi_i$.*

Proof: We prove by induction on i in ascending order.

$\boxed{\text{Base case}}$

For $i = 1$, the result follows trivially from the definition of $Ok_1(S)$.

$\boxed{\text{Inductive case}}$

Pick $P_i\rho \wedge \ddot{\theta} \Rightarrow \ddot{\theta}' \in \Phi_i$. It suffices to show that $S(P_i) \Rightarrow S(\ddot{\theta} \Rightarrow \ddot{\theta}')\rho^{-1}$.

Let $k < i$ be the largest k such that P_k appears in $\ddot{\theta}$. Let $P_k\rho_k \wedge \ddot{\theta}_k = \ddot{\theta}$. The following is a tautology.

$$S(P_i)\rho \Rightarrow ((S(\ddot{\theta}) \wedge (S(P_i\rho \wedge \ddot{\theta}_k) \Rightarrow S(\ddot{\theta}')) \Rightarrow S(\ddot{\theta}'))$$

By induction hypothesis, we also have $S(P_k)\rho_k \Rightarrow (S(P_i\rho \wedge \ddot{\theta}_k) \Rightarrow S(\ddot{\theta}'))$. Therefore, $S(P_i) \Rightarrow S(\ddot{\theta} \Rightarrow \ddot{\theta}')\rho^{-1}$.

Now, suppose no such $k < i$ exists. Then, we have $S(P_i) \Rightarrow S(\ddot{\theta} \Rightarrow \ddot{\theta}')\rho^{-1}$ by the definition of $Ok_1(S)$.

□

Finally, Theorem 5.8 follows from Lemma C.12 and Lemma C.13.