Predicate Abstraction and CEGAR for Higher-Order Model Checking

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

Ryosuke Sato
Tohoku University
ryosuke@kb.ecei.tohoku.ac.jp

Hiroshi Unno
Tohoku University
uhiro@kb.ecei.tohoku.ac.jp

[1]

Abstract

Higher-order model checking (more precisely, the model checking of higher-order recursion schemes) has been extensively studied recently, which can automatically decide properties of programs written in the simply-typed λ-calculus with recursion and *finite* data domains. This paper formalizes predicate abstraction and counterexample-guided abstraction refinement (CEGAR) for higher-order model checking, enabling automatic verification of programs that use *infinite* data domains such as integers. A prototype verifier for higher-order functional programs based on the formalization has been implemented and tested for several programs.

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

General Terms Languages, Reliability, Verification

Keywords Predicate Abstraction, CEGAR, Higher-Order Model Checking, Dependent Types

1. Introduction

The model checking of higher-order recursion schemes (*recursion schemes*, for short) has been extensively studied [19, 24, 28], and recently applied to verification of functional programs [20, 22, 26]. Recursion schemes are grammars for describing infinite trees [19, 28], and the recursion scheme model checking is concerned about whether the tree generated by a recursion scheme satisfies a given property. It can be considered an extension of finite state and pushdown model checking, where the model checking of order-0 and order-1 recursion schemes respectively correspond to finite state and pushdown model checking. From a programming language point of view, a recursion scheme is a term of the simply-typed, call-by-name λ-calculus with recursion and tree constructors, which generates a single, possibly infinite tree. Various verification problems for functional programs can be easily reduced to recursion scheme model checking problems [20, 22, 26]. Thanks to the decidability of recursion scheme model checking [28], the reduction yields a sound, complete, and automatic verification method for programs written in the simply-typed λ-calculus with recursion and *finite* data domains (such as booleans).

There is, however, still a large gap between the programs handled by the above-mentioned method and real functional programs. One of the main limitations is that infinite data domains such as integers and lists cannot be handled by the recursion scheme model checking. To overcome that limitation, this paper extends the techniques of predicate abstraction [12] and counterexample-guided abstraction refinement (CEGAR) [4, 8] for higher-order model checking (i.e., recursion scheme model checking).

The overall structure of our method is shown in Figure 1. Given a higher-order functional program, predicate abstraction is first applied to obtain a higher-order boolean program (Step 1 in Figure 1). For example, consider the following program M_1:

```
let f x g = g(x+1) in let h y = assert(y>0) in
let k n = if n>0 then f n h else () in k(randi())
```

Here, `assert` takes a boolean as an argument and is reduced to `fail` if the argument is false. The function `randi` returns a non-deterministic integer value. Using a predicate $\lambda x.x > 0$, we obtain the following higher-order boolean program e_1:

```
let f b g = if b then g(true) else g(randb()) in
let h c = assert(c) in
let k () = if randb() then f true h else () in k()
```

Here, `randb` returns a non-deterministic boolean value. Note that the integer variables `x` and `y` have been replaced by the boolean variable `b` and `c` respectively, which represents whether the values of `x` and `y` are greater than `0`. In the abstract version of `f`, `b` being true means that `x>0`, which implies `x+1>0`, so that `true` is passed to `g` in the then-part. In the else-part, `x<=0`, hence `x+1>0` may or may not hold, so that a non-deterministic boolean value is passed to `g`. The higher-order boolean program thus obtained is an abstraction of the source program; for any reduction sequence of the source program, there is a corresponding reduction sequence of the higher-order boolean program (but not vice versa). Thus, for example, if the abstract program does not cause an assertion failure, neither does the source program.

The higher-order boolean program is then represented as a recursion scheme and model-checked by using an existing recursion scheme model checker [21, 22] (Step 2 in Figure 1). If the higher-order boolean program satisfies a given safety property,[2] the source program is also safe. Otherwise, an error path of the boolean program is inspected (Step 3 in Figure 1). If it is also an error path of the source program, then it is reported that the program is unsafe. Otherwise, new predicates are extracted from the error path, in order to refine predicate abstraction (Step 4 in Figure 1).

In the example above, we actually start predicate abstraction with the empty set of predicates, and obtain the following abstract program e_0:

```
let f g = g() in let h () = assert(randb()) in
```

[1] © ACM, 2011. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in Proceedings of PLDI 2011.

[2] For the sake of simplicity, throughout the paper, we only consider the reachability property.

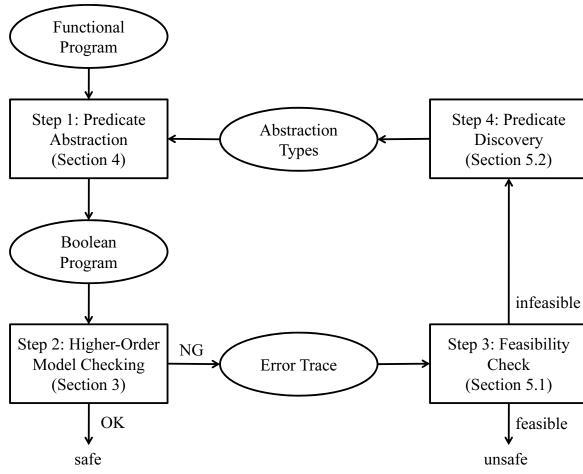

Figure 1. Higher-Order Model Checking with Predicate Abstraction and CEGAR

```
let k () = if randb() then f h else () in k()
```

The model checking of this program yields the following reduction sequence, leading to an assertion failure:

$$\begin{array}{l}\texttt{k()} \longrightarrow \texttt{if randb() then f h else ()} \\ \longrightarrow \texttt{if true then f h else ()} \longrightarrow \texttt{f h} \\ \longrightarrow \texttt{h()} \longrightarrow \texttt{assert}\,(\texttt{randb()}) \\ \longrightarrow \texttt{assert}\,(\texttt{false}) \longrightarrow \texttt{fail} \qquad \cdots (1)\end{array}$$

The corresponding reduction sequence in the source program M_1 is:

$$\begin{array}{l}\texttt{k n} \longrightarrow \texttt{if n>0 then f n h else ()} \longrightarrow_{\texttt{n>0}} \texttt{f n h} \\ \longrightarrow \texttt{h(n+1)} \longrightarrow \texttt{assert (n+1>0)} \longrightarrow_{\texttt{n+1<=0}} \texttt{fail} \quad \cdots (2)\end{array}$$

Here, n is some integer, and we have annotated the sequence with the conditions that should hold at each step. As $\texttt{n>0} \wedge \texttt{n+1<=0}$ is unsatisfiable, we know that the reduction sequence above is actually infeasible, so that the source program may not cause an assertion failure. From the unsatisfiable constraint above, we can learn that information about whether an integer is positive is useful. By using it, we get the refined abstract program shown earlier. As the new abstract program is safe (i.e. does not cause an assertion failure), we can conclude that the source program is also safe.

The idea sketched above is basically the same as the techniques for predicate abstraction and CEGAR used already in finite state and pushdown model checking [4, 8], except that models have been replaced by higher-order boolean programs (or recursion schemes). As discussed below, however, it turned out that there are many challenging problems in developing effective methods for predicate abstraction and CEGAR for higher-order model checking.

First, for predicate abstraction, it is unreasonable to use the same set of predicates for all the integer variables. For example, let us modify the program above into the following program M_2:

```
let f x g = g(x+1) in let h y = assert(y>0) in
let k n = if n>=0 then f n h else () in k(randi())
```

Then, the predicate $\lambda\nu.\nu \geq 0$ should be used for x, while $\lambda\nu.\nu > 0$ should be used for y. We should consistently use predicates; for example, with the choice of the predicates above, g's argument should be abstracted by using $\lambda\nu.\nu > 0$, rather than $\lambda\nu.\nu \geq 0$. We use *types* (called *abstraction types*) to express which predicate should be used for each variable. For example, for the above program, the following abstraction types are assigned to f, h, and k:

$$\begin{array}{l} f : \texttt{int}[\lambda\nu.\nu \geq 0] \rightarrow (\texttt{int}[\lambda\nu.\nu > 0] \rightarrow \star) \rightarrow \star \\ h : \texttt{int}[\lambda\nu.\nu > 0] \rightarrow \star \qquad k : \texttt{int}[\,] \rightarrow \star \end{array}$$

The type of f means that the first argument of f should be an integer abstracted by the predicate $\lambda\nu.\nu \geq 0$, and the second argument be a function that takes an integer abstracted by the predicate $\lambda\nu.\nu > 0$ as an argument and returns a unit value.[3] By using these abstraction types, the problem of checking that predicates are consistently used boils down to a type checking problem. For example, the standard rule for application:

$$\frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

ensures that N is abstracted using the predicates expected by the function M; there is no such case that an abstraction of function M expects a value abstracted by using the predicate $\lambda\nu.\nu > 0$ but the actual argument N is abstracted by using $\lambda\nu.\nu \geq 0$.

A further twist is necessary to deal with multi-ary predicates. For example, consider the following modified version M_3:

```
let f x g = g(x+1) in let h z y = assert(y>z) in
let k n = if n>=0 then f n (h n) else () in
  k(randi())
```

The variable y should now be abstracted by using $\lambda\nu.\nu > \texttt{z}$, which depends on the value of z. Thus, the above program should be abstracted by using the following *dependent* abstraction types:

$$\begin{array}{l} f : (x : \texttt{int}[\,] \rightarrow (w : \texttt{int}[\lambda\nu.\nu > x] \rightarrow \star) \rightarrow \star) \\ h : (z : \texttt{int}[\,] \rightarrow y : \texttt{int}[\lambda\nu.\nu > z] \rightarrow \star) \qquad k : \texttt{int}[\,] \rightarrow \star \end{array}$$

Here, please note that the types of the second arguments of f and h refer to the values of the first arguments. Thus, our type system for ensuring the consistency of predicates is actually a *dependent* one. A predicate abstraction algorithm is then formalized as a type-directed transformation relation $\Gamma \vdash M : \tau \Rightarrow e$ based on the dependent abstraction type system, where M is a source program and e is an abstract program.[4]

The predicate abstraction mentioned above is sound in the sense that if an abstract program is safe (i.e., does not reach fail), so is the source program. Further, we can show that it is relatively complete with respect to a dependent (refinement) intersection type system [32]: If a source program is typable in the dependent intersection type system, our predicate abstraction can generate a safe abstract boolean program by using certain abstraction types. This means that, as long as suitable predicates are provided (by a user or an automated method like the CEGAR discussed below), the combination of our predicate abstraction and higher-order model checking has at least the same verification power as (and actually strictly more expressive than, as discussed later: see Remark 1 in Section 4) the dependent intersection type system. Here, note that we need only atomic predicates used in the dependent types; higher-order model checking can look for arbitrary boolean combinations of the atomic predicates as candidates of dependent types. Thus, this part alone provides a good alternative to Liquid types [31], which also asks users to provide templates of predicates, and infers dependent types. Thanks to the power of higher-order model checking, however, our technique can infer dependent, *intersection* types unlike Liquid types.

[3] Here, abstraction types should not be confused with refinement types [35]; the abstraction type of a term only tells how the term should be abstracted, not what are possible values of the term. For example, integer 3 can have type $\texttt{int}[\lambda\nu.\nu < 0]$ (and it will be abstracted to the boolean value *false*).

[4] To avoid the confusion, we call dependent abstraction types just *abstraction types* below. We use the term "dependent types" to refer to ordinary dependent types used for expressing refinement of simple types.

We now discuss the CEGAR part. Given an error path of an abstract boolean program, we can find a corresponding (possibly infeasible) error path of the source program. Whether the error path is feasible in the source program can be easily decided by symbolically executing the source program along the error path, and checking whether all the branching conditions in the path are satisfiable (recall the example given earlier). The main question is, if the error path turns out to be infeasible, how to find a suitable refinement of abstraction types, so that the new abstraction types yield an abstract boolean program that does not contain the infeasible error path. This has been well studied for first-order programs [2–4, 8, 13–15], but it is not clear how to lift those techniques to deal with higher-order programs.

Our approach to finding suitable abstraction types is as follows. From a source program and its infeasible error path, we first construct a *straightline higher-order program* (abbreviated to SHP) that exactly corresponds to the infeasible path, and contains neither recursion nor conditional branches. In the case of the program M_3 above, this is easily obtained, as follows:

```
let f1 x g = g(x+1) in let h1 z y = assert(y>z) in
let k1 n = assume(n>=0); f1 n (h1 n) in k1(c)
```

Here, `c` is a constant, and `assume`(b) evaluates b, and proceeds to the next instruction only if b is true. (But unlike `assert`, it is not reduced to `fail` even if b is false.) For general programs that contain recursions, the construction is more involved: see Section 5.

For SHP, a standard dependent (refinement) type system is sound and *complete*, in the sense that a program does not reach `fail` if and only if the program is typable in the type system. Further, (a sub-procedure of) previous algorithms for inferring dependent types based on interpolants [32, 33] is actually complete (modulo the assumption that the underlying logic is decidable and interpolants can always be computed) for SHP. Thus, we can automatically infer the dependent type of each function in the straightline program. For example, for the program above, we obtain:

$$\begin{array}{l} \texttt{f1} : (x:\texttt{int} \rightarrow (y:\{\nu:\texttt{int} \mid \nu > x\} \rightarrow \star) \rightarrow \star) \\ \texttt{h1} : (z:\texttt{int} \rightarrow y:\{\nu:\texttt{int} \mid \nu > z\} \rightarrow \star) \\ \texttt{k1} : (z:\texttt{int} \rightarrow \star) \end{array}$$

Here, the type of `h1` means that given integers z and y such that $y > z$, $\texttt{h1}\, z\, y$ returns a unit value without reaching `fail`. (These dependent types should not be confused with abstraction types: the latter only provides information about how the source program should be abstracted.)

We then refine the abstraction type of each function in the source program with predicates occurring in the dependent types of the corresponding functions in the SHP. For example, given the above dependent types, we get the following abstraction types:

$$\begin{array}{l} \texttt{f} : (x:\texttt{int}[\,] \rightarrow (y:\texttt{int}[\lambda\nu.\nu > x] \rightarrow \star) \rightarrow \star) \\ \texttt{h} : (z:\texttt{int}[\,] \rightarrow y:\texttt{int}[\lambda\nu.\nu > z] \rightarrow \star) \qquad \texttt{k} : \texttt{int}[\,] \rightarrow \star \end{array}$$

We can show that the abstraction types inferred in this manner are precise enough, in that the abstract program obtained by using the new abstraction types no longer has the infeasible error path. (Thus, the so-called "progress" property is guaranteed as in CEGAR methods for finite-state or pushdown model checking.)

Based on the predicate abstraction and CEGAR techniques mentioned above, we have implemented a prototype verifier for (simply-typed) higher-order functional programs with recursion and integer base types, and tested it for several programs.

Our contributions include: (i) the formalization of predicate abstraction for higher-order programs, based on the novel notion of abstraction types, (ii) the formalization of CEGAR for higher-order programs, based on the novel notion of SHP and reduction of the predicate discovery problem to dependent type inference, (iii) theoretical properties like the relative completeness of our method with respect to a dependent intersection type, the progress property, etc., and (iv) the implementation and preliminary experiments.

The rest of this paper is structured as follows. Section 2 introduces the source language, and Section 3 introduces a language of higher-order boolean programs, and reviews the result on higher-order model checking. Sections 4 and 5 respectively formalize predicate abstraction and CEGAR for higher-order programs. Section 6 reports our prototype implementation and preliminary experiments. Section 7 discusses related work, and Section 8 concludes. For the space restriction, proofs are omitted, which are available in a full version [25].

2. Language

This section introduces a simply-typed, higher-order functional language, which is used as the target of our verification method.

We assume a set $\mathbf{B} = \{b_1, \ldots, b_n\}$ of data types, and a set $[\![b_i]\!]$ of constants, ranged over by c, for each data type b_i. We also assume that there are operators $\mathrm{op} : b_{i_1}, \ldots, b_{i_k} \rightarrow b_j$; we write $[\![\mathrm{op}]\!]$ for the function denoted by op. We assume that the set of data types includes $\star$ with $[\![\star]\!] = \langle\rangle$, and `bool` with $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$, and that the set of operators includes $= : b, b \rightarrow \texttt{bool}$ for every $b \in \mathbf{B}$, and boolean connectives such as $\wedge : \texttt{bool}, \texttt{bool} \rightarrow \texttt{bool}$.

The syntax of the language is given by:

$$\begin{array}{rcl} D \text{ (program)} & ::= & \{f_1\, \widetilde{x}_1 = e_1, \ldots, f_m\, \widetilde{x}_m = e_m\} \\ e \text{ (expressions)} & ::= & c \mid x\, \widetilde{v} \mid f\, \widetilde{v} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \mathrm{op}(\widetilde{v}) \\ & \mid & \texttt{fail} \mid \texttt{assume}\ v; e \mid e_1 \,\Box\, e_2 \\ v \text{ (values)} & ::= & c \mid x \mid f\, \widetilde{v} \end{array}$$

Here, $\widetilde{x}$ abbreviates a (possibly empty) sequence $x_1, \ldots, x_n$. In the definition $f\, \widetilde{x} = e$, we call the length of $\widetilde{x}$ the *arity* of f. In the definition of values, the length of $\widetilde{v}$ in $f\, \widetilde{v}$ must be smaller than the arity of f. (In other words, $f\, \widetilde{v}$ must be a partial application.) We assume that every function in D has a non-zero arity, and that D contains a distinguished function symbol $main \in \{f_1, \ldots, f_m\}$ whose simple type is $\star \rightarrow \star$.

Most of the expressions are standard, except the following ones. The expression `fail` aborts the program and reports a failure. The expression `assume` $v; e$ evaluates e if v is true; otherwise it stops the program (without a failure). The expression $e_1 \,\Box\, e_2$ evaluates e_1 or e_2 in a non-deterministic manner. Note that a standard conditional expression `if` v `then` e_1 `else` e_2 can be expressed as:

$$(\texttt{assume}\ v; e_1) \,\Box\, (\texttt{let}\ x = \neg v\ \texttt{in}\ \texttt{assume}\ x; e_2).$$

We can express the assertion `assert` v as `if` v `then` $\langle\rangle$ `else` `fail`. The random number generator *randi* used in Section 1 is defined by:

$$\begin{array}{l} \textit{randi}\ \langle\rangle = (\textit{randiFrom}\ 1)\Box(\textit{randiTo}\ 0) \\ \textit{randiFrom}\ n = n\Box(\textit{randiFrom}\ (n+1)) \\ \textit{randiTo}\ n = n\Box(\textit{randiTo}\ (n-1)) \end{array}$$

We assume that a program is well-typed in the standard simple type system, where the set of types is given by:

$$\tau ::= b_1 \mid \cdots \mid b_n \mid \tau_1 \rightarrow \tau_2.$$

Furthermore, we assume that the body of each definition has a data type b_i, not a function type. This is not a limitation, as we can always use the continuation-passing-style (CPS) transformation to transform a higher-order program to an equivalent one that satisfies the restriction.

We define the set of *evaluation contexts* by: $E ::= [\,] \mid \texttt{let}\ x = E\ \texttt{in}\ e$. The reduction relation is given in Figure 2. We label the reduction relation with $0, 1, \epsilon$ to record which branch has been chosen by a non-deterministic expression $e_1 \,\Box\, e_2$; it will be used

$$\frac{f\,\widetilde{x} = e \in D}{E[f\,\widetilde{v}] \xrightarrow{\epsilon}_D E[[\widetilde{v}/\widetilde{x}]e]} \quad \text{(E-App)}$$

$$E[\texttt{let}\ x = v\ \texttt{in}\ e] \xrightarrow{\epsilon}_D E[[v/x]e] \quad \text{(E-Let)}$$

$$E[\mathrm{op}(\widetilde{c})] \xrightarrow{\epsilon}_D E[[\![\mathrm{op}]\!](\widetilde{c})] \quad \text{(E-Op)}$$

$$E[e_0 \,\square\, e_1] \xrightarrow{i}_D E[e_i] \quad \text{(E-Par)}$$

$$E[\texttt{assume true}; e] \xrightarrow{\epsilon}_D E[e] \quad \text{(E-Assume)}$$

$$E[\texttt{fail}] \xrightarrow{\epsilon}_D \texttt{fail} \quad \text{(E-Fail)}$$

Figure 2. Call-by-Value Operational Semantics

to relate reductions of a source program and an abstract program later in Sections 4 and 5. We write $e_1 \overset{l_1 \cdots l_n}{\Longrightarrow}_D e_2$ if

$$e_1 (\xrightarrow{\epsilon}_D)^* \xrightarrow{l_1}_D (\xrightarrow{\epsilon}_D)^* \cdots (\xrightarrow{\epsilon}_D)^* \xrightarrow{l_n}_D (\xrightarrow{\epsilon}_D)^* e_2.$$

We often omit the subscript D when it is clear from the context. The goal of our verification method is to check whether $\mathit{main}\,\langle\rangle \overset{s}{\Longrightarrow}_D \texttt{fail}$.[5]

3. Higher-Order Boolean Programs and Model Checking

A source program is translated to a higher-order boolean program (abbreviated to HBP) by the predicate abstraction discussed in Section 4. The language of HBP is essentially the same as the source language in the previous section, except:

- The set of data types consists only of types of the form $\underbrace{\texttt{bool} \times \cdots \times \texttt{bool}}_{m}$ (which is identified with $\star$ when $m = 0$, and $\texttt{bool}$ when $m = 1$). We assume there are the following operators to construct or deconstruct tuples:

$$\begin{array}{rcl} \langle \cdot, \ldots, \cdot \rangle & : & \texttt{bool}, \ldots, \texttt{bool} \to \texttt{bool} \times \cdots \times \texttt{bool} \\ \sharp_i & : & \texttt{bool} \times \cdots \times \texttt{bool} \to \texttt{bool} \end{array}$$

- The set of expressions is extended with $e_1 \blacksquare e_2$ and unnamed functions $\lambda x.e$. The former is used for expressing the non-determinism introduced by abstractions; it is the same as $e_1 \,\square\, e_2$, which is used to express the non-determinism present in a source program, except that the reduction is labelled with ϵ. This distinction is convenient for the CEGAR procedure discussed in Section 5 to find a corresponding execution path of the source program from an execution path of the abstract program. Unnamed functions are used just for technical convenience for defining predicate abstraction; with λ-lifting, we can easily get rid of λ-abstractions. (The evaluation rules and evaluation contexts are accordingly extended with $E[(\lambda x.e)v] \longrightarrow E[[v/x]e]$ and $E ::= \cdots \mid E\,e \mid v\,E$.)

The following theorem is the basis of our verification method. It follows immediately from the decidability of the model checking of higher-order recursion schemes [28].

Theorem 3.1. *Let D be an HBP. The property $\exists s.(\mathit{main}\langle\rangle \overset{s}{\Longrightarrow}_D \texttt{fail})$ is decidable.*

[5] Thus, we consider the reachability problem for a closed program. Note, however, that we can express unknown values by using non-determinism (recall *randi*). It is also easy to extend our method to deal with more general verification problems, such as resource usage verification [22].

$$\frac{\texttt{A2S}(\Gamma), x : b \vdash_{\texttt{ST}} \psi_i : \texttt{bool} \text{ for each } i \in \{1, \ldots, n\}}{\Gamma \vdash_{\texttt{wf}} b[\lambda x.\psi_1, \ldots, \lambda x.\psi_n]}$$

$$\frac{\Gamma \vdash_{\texttt{wf}} \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash_{\texttt{wf}} \sigma_2}{\Gamma \vdash_{\texttt{wf}} x : \sigma_1 \to \sigma_2} \qquad \frac{}{\vdash_{\texttt{wf}} \emptyset} \qquad \frac{\vdash_{\texttt{wf}} \Gamma \qquad \Gamma \vdash_{\texttt{wf}} \sigma}{\vdash_{\texttt{wf}} \Gamma, x : \sigma}$$

Figure 3. Well-formed types and type environments

We can use a recursion scheme model checker TRecS [21, 22] to decide the above property.[6] If $\exists s.(\mathit{main}\langle\rangle \overset{s}{\Longrightarrow}_D \texttt{fail})$ holds, the model checker generates an error path s. The knowledge about recursion schemes is unnecessary for understanding the rest of this paper, but an interested reader may wish to consult [20, 22, 28].

4. Predicate Abstraction

This section formalizes predicate abstraction for higher-order programs. As explained in Section 1, we use *abstraction types* to express which predicates should be used for abstracting each subexpression. The syntax of abstraction types is given by:

$$\begin{array}{rcl} \sigma \text{ (abstraction types)} & ::= & b_1[\widetilde{P}] \mid \cdots \mid b_n[\widetilde{P}] \mid x : \sigma_1 \to \sigma_2 \\ P, Q \text{ (predicates)} & ::= & \lambda x.\psi \\ \Gamma \text{ (type environments)} & ::= & \emptyset \mid \Gamma, f : \sigma \mid \Gamma, x : \sigma \end{array}$$

Here, the meta-variable ψ represents an expression of type $\texttt{bool}$ (which is called *a formula*) that is constructed only from variables of base types, constants, and primitive operations; we do not allow formulas that contain function applications, like $y > f(x)$. The base type $b_i[\widetilde{P}]$ describes values v of type b_i that should be abstracted to a tuple of booleans $\langle P_1(v), \ldots, P_n(v) \rangle$. For example, the integer 3 with the abstraction type $\texttt{int}[\lambda\nu.\nu > 0, \lambda\nu.\nu < 2]$ is abstracted to $\langle \texttt{true}, \texttt{false} \rangle$. We often abbreviate $b[\,]$ to b. The dependent function type $x\!:\!\sigma_1 \to \sigma_2$ describes functions that take a value v of type σ_1, and return a value of type $[v/x]\sigma_2$. The scope of x in the type $x\!:\!\sigma_1 \to \sigma_2$ is σ_2. When x does not occur in σ_2, we often write $\sigma_1 \to \sigma_2$ for $x\!:\!\sigma_1 \to \sigma_2$. As mentioned already, abstraction types only describe how each expression should be abstracted, not the actual value. For example, 3 can have type $\texttt{int}[\lambda\nu.\nu < 0]$, and $\lambda x.x$ can have type $x : \texttt{int}[\,] \to \texttt{int}[\lambda\nu.\nu = x + 1]$ (and abstracted to $\lambda x : \star.\texttt{false}$).

We do not consider types whose predicates are ill-typed or violate a variable's scope, such as $x\!:\!\texttt{bool}[\,] \to \texttt{int}[\lambda y.x\!+\!1 = y]$ and $x : \texttt{int}[\lambda x.y > x] \to y : \texttt{int}[\,] \to \texttt{bool}[\,]$. (The former uses a boolean variable as an integer, and the latter refers to the variable y outside its scope.) Figure 3 defines the well-formedness conditions for types and type environments. In the figure, $\Delta \vdash_{\texttt{ST}} e : \tau$ denotes the type judgment of the standard simple type system. We write $\texttt{A2S}(\sigma)$ and $\texttt{A2S}(\Gamma)$ respectively for the simple type and the simple type environment obtained by removing predicates. For example, $\texttt{A2S}(f : (x : \texttt{int}[\,] \to \texttt{int}[\lambda y.y > x]), z : \texttt{int}[\lambda z.z > 0]) = f : \texttt{int} \to \texttt{int}, z : \texttt{int}$.

Figure 4 defines the predicate abstraction relation $\Gamma \vdash e_1 : \sigma \rightsquigarrow e_2$, which reads that an expression e_1 (of the source language in Section 2) can be abstracted to an expression e_2 (of the HBP language given in Section 3) by using the abstraction type σ, under the assumption that each free variable x of e_1 has been abstracted using the abstraction type $\Gamma(x)$. In the rules, it is implicitly assumed that all the type environments and types are well-formed. We do not distinguish between function vari-

[6] The gap between the operational semantics of our language and that of recursion schemes can be filled by the CPS transformation. Note also that finite state or pushdown model checkers cannot be used, as higher-order programs are in general strictly more expressive [10].

$$\frac{\begin{array}{c} e \text{ is a constant, a variable or an expression of the form } \mathrm{op}(\widetilde{v}) \\ \mathtt{A2S}(\Gamma) \vdash_{\mathtt{ST}} e : b \end{array}}{\Gamma \vdash e : b[\lambda\nu.\nu = e] \leadsto \mathtt{true}} \quad \text{(A-BASE)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : b[\widetilde{Q}] \leadsto e' \\ \beta(\Gamma, x : b[\widetilde{Q}]) \vdash_{\mathtt{ST}} \psi : \mathtt{bool} \qquad \beta(\Gamma, x : b[\widetilde{Q}]) \vdash_{\mathtt{ST}} \psi' : \mathtt{bool} \\ \models P(x) \Rightarrow \theta_{\Gamma, x:b[\widetilde{Q}]}(\psi) \qquad \models \neg P(x) \Rightarrow \theta_{\Gamma, x:b[\widetilde{Q}]}(\psi') \\ e'' = (\mathtt{assume}\ \psi; \mathtt{true}) \blacksquare (\mathtt{assume}\ \psi'; \mathtt{false}) \end{array}}{\Gamma \vdash e : b[\widetilde{Q}, P] \leadsto \mathtt{let}\ x = e'\ \mathtt{in}\ \langle x, e'' \rangle} \quad \text{(A-CADD)}$$

$$\frac{\Gamma \vdash e : b[\widetilde{Q}, \widetilde{P}] \leadsto e'}{\Gamma \vdash e : b[\widetilde{P}] \leadsto \mathtt{let}\ \langle \widetilde{x}, \widetilde{y} \rangle = e'\ \mathtt{in}\ \langle \widetilde{y} \rangle} \quad \text{(A-CREM)}$$

$$\frac{\begin{array}{c} \Gamma(x) = (y_1 : \sigma_1 \to \cdots \to y_k : \sigma_k \to \sigma) \\ \Gamma, y_1 : \sigma_1, \ldots, y_{i-1} : \sigma_{i-1} \vdash v_i : [v_1/y_1, \ldots, v_{i-1}/y_{i-1}]\sigma_i \leadsto e_i \\ \text{(for each } i \in \{1, \ldots, k\}) \end{array}}{\Gamma \vdash x\ \widetilde{v} : [\widetilde{v}/\widetilde{y}]\sigma \leadsto \begin{array}{l} \mathtt{let}\ y_1 = e_1\ \mathtt{in}\ \cdots \\ \mathtt{let}\ y_k = e_k\ \mathtt{in}\ x\ \widetilde{y} \end{array}} \quad \text{(A-APP)}$$

$$\frac{\Gamma \vdash e_1 : \sigma' \leadsto e_1' \qquad \Gamma, x : \sigma' \vdash e_2 : \sigma \leadsto e_2'}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \sigma \leadsto \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2'} \quad \text{(A-LET)}$$

$$\frac{\beta(\Gamma) \vdash_{\mathtt{ST}} \psi : \mathtt{bool} \qquad \models \theta_\Gamma(\psi)}{\Gamma \vdash \mathtt{fail} : \sigma \leadsto \mathtt{assume}\ \psi; \mathtt{fail}} \quad \text{(A-FAIL)}$$

$$\frac{\begin{array}{c} \Gamma \vdash v : \mathtt{bool}[\lambda x.x] \leadsto e_1 \\ \Gamma, x : \mathtt{bool}[\lambda x.v] \vdash e : \sigma \leadsto e_2 \end{array}}{\Gamma \vdash \mathtt{assume}\ v; e : \sigma \leadsto \mathtt{let}\ x = e_1\ \mathtt{in}\ \mathtt{assume}\ x; e_2} \quad \text{(A-ASM)}$$

$$\frac{\Gamma \vdash e_1 : \sigma \leadsto e_1' \qquad \Gamma \vdash e_2 : \sigma \leadsto e_2'}{\Gamma \vdash e_1 \,\Box\, e_2 : \sigma \leadsto e_1' \,\Box\, e_2'} \quad \text{(A-PAR)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : (x : \sigma_1' \to \sigma_2') \leadsto e' \qquad \Gamma, x : \sigma_1 \vdash x : \sigma_1' \leadsto e_1' \\ \Gamma, x : \sigma_1, x' : \sigma_1', y : \sigma_2' \vdash y : \sigma_2 \leadsto e_2' \end{array}}{\Gamma \vdash e : (x : \sigma_1 \to \sigma_2) \leadsto \begin{array}{l} \mathtt{let}\ f = e'\ \mathtt{in} \\ \lambda x.\mathtt{let}\ x' = e_1'\ \mathtt{in} \\ \quad \mathtt{let}\ y = f\ x'\ \mathtt{in}\ e_2' \end{array}} \quad \text{(A-CFUN)}$$

$$\frac{\begin{array}{c} f_i : (\widetilde{x}_i : \widetilde{\sigma}_i \to \sigma_i) \in \Gamma \qquad \Gamma, \widetilde{x}_i : \widetilde{\sigma}_i \vdash e_i : \sigma_i \leadsto e_i' \\ \text{(for each } i \in \{1, \ldots, m\}) \qquad \Gamma(\mathit{main}) = \star[\,] \to \star[\,] \end{array}}{\begin{array}{c} \vdash \{f_1\ \widetilde{x}_1 = e_1, \ldots, f_m\ \widetilde{x}_m = e_m\} : \Gamma \leadsto \\ \{f_1\ \widetilde{x}_1 = e_1', \ldots, f_m\ \widetilde{x}_m = e_m'\} \end{array}} \quad \text{(A-PROG)}$$

Figure 4. Predicate Abstraction Rules

ables and other variables (hence, A-APP applies also to a function variable f). In A-CADD, $\mathtt{assume}\ e_1; e_2$ is a syntax sugar for $\mathtt{let}\ x_1 = e_1\ \mathtt{in}\ \mathtt{assume}\ x_1; e_2$.

In A-CADD and A-FAIL, $\beta(\sigma)$ ($\beta(\Gamma)$, resp.) represent the simple type (simple type environment, resp.) obtained by replacing each occurrence of a base abstraction type $b[P_1, \ldots, P_m]$ with $\underbrace{\mathtt{bool} \times \cdots \times \mathtt{bool}}_{m}$. Intuitively, $\beta(\Gamma)$ represents the type environment for the output program of the transformation.

We explain the main rules. Base values are abstracted by using three rules A-BASE, A-CADD, and A-CREM. Before explaining those rules, let us discuss the following simplified version, specialized for a single predicate:

$$\frac{\models P(e) \Rightarrow \theta_\Gamma(\psi) \qquad \models \neg P(e) \Rightarrow \theta_\Gamma(\psi')}{\Gamma \vdash e : b[P] \leadsto (\mathtt{assume}\ \psi; \mathtt{true}) \blacksquare (\mathtt{assume}\ \psi'; \mathtt{false})} \quad \text{(A-BSIMP)}$$

Here, we assume that e is a constant, a variable, or an expression of the form $\mathrm{op}(\widetilde{v})$ and has a base type b. ψ and ψ' are boolean formulas that may contain variables in Γ. As e may contain variables, we need to take into account information about the values of the variables, which is obtained by using the substitution θ_Γ, defined as: $\{x \mapsto \langle P_1(x), \ldots, P_m(x) \rangle \mid \Gamma(x) = b[P_1, \ldots, P_m]\}$. For example, let Γ be $x : \mathtt{int}[\lambda x.x > 0, \lambda x.x < 0]$ and ψ be $\sharp_1(x) \wedge \sharp_2(x)$. Then, $\theta_\Gamma(\psi) = x > 0 \wedge x < 0$. As in this example, the substitution θ_Γ maps a boolean expression of an abstract program to the corresponding condition in the source program. In rule A-BSIMP above, $\models P(e) \Rightarrow \theta_\Gamma(\psi)$ means that $P(e)$ is true only if $\theta_\Gamma(\psi)$ is true, i.e. the value of ψ in the abstract program is true. Thus, the abstract value of e may be $\mathtt{true}$ only if the value of ψ is $\mathtt{true}$, hence the part $\mathtt{assume}\ \psi; \mathtt{true}$ in the abstract program. Similarly, the abstract value of e may be $\mathtt{false}$ only if the value of ψ' is $\mathtt{true}$, hence the part $\mathtt{assume}\ \psi'; \mathtt{false}$.

For example, let $e \equiv x + 1$, $P \equiv \lambda x \geq 0$, and $\Gamma \equiv x : \mathtt{int}[P]$. Then, $\models P(x+1) \Rightarrow \mathtt{true}$ and $\models \neg P(x+1) \Rightarrow \neg P(x)$, so that e is abstracted to $(\mathtt{assume}\ \mathtt{true}; \mathtt{true}) \blacksquare (\mathtt{assume}\ \neg x; \mathtt{false})$. Note that $\theta_\Gamma(\neg x) = [P(x)/x]\neg x = \neg P(x)$.

We need to generalize the above rule to the case for multiple predicates. The following is a naive rule.

$$\frac{\Gamma \vdash e : b[P_i] \leadsto e_i}{\Gamma \vdash e : b[P_1, \ldots, P_n] \leadsto \langle e_1, \ldots, e_n \rangle} \quad \text{(A-BCARTESIAN)}$$

This produces a well-known cartesian abstraction, which is often too imprecise. The problem is that each boolean value of the abstraction is computed separately, ignoring the correlation. For example, let $P_1 \equiv \lambda x.x > 0$ and $P_2 \equiv \lambda x.x \leq 0$ with $n = 2$. Then, a possible abstraction of an unknown integer should be $\langle \mathtt{true}, \mathtt{false} \rangle$ and $\langle \mathtt{false}, \mathtt{true} \rangle$, but the above rule would generate $\langle (\mathtt{true} \blacksquare \mathtt{false}), (\mathtt{true} \blacksquare \mathtt{false}) \rangle$, which also contains $\langle \mathtt{true}, \mathtt{true} \rangle$ and $\langle \mathtt{false}, \mathtt{false} \rangle$.

The discussion above motivated us to introduce the three rules A-BASE, A-CADD, and A-CREM. In order to abstract an expression e with $b[P_1, \ldots, P_n]$, we first use A-BASE to abstract e to $\mathtt{true}$ by using the abstraction type $b[\lambda\nu.\nu = e]$; this is necessary to keep the exact information about e during the computation of abstractions. A-CADD is then used to add predicates $P_1, \ldots, P_n$ one by one, taking into account the correlation between the predicates. Note that in A-CADD, the result of abstraction by the other predicates is taken into account by the substitution $\theta_{\Gamma, x:b[\widetilde{Q}]}$. Finally, A-CREM is used to remove the unnecessary predicate $\lambda\nu.\nu = e$. See Example 4.1 for an application of these rules.

Note that rule A-CADD is non-deterministic in the choice of conditions ψ and ψ', so that how to compute the conditions is left unspecified. We have intentionally made so, because depending on base data types, the most precise conditions (the strongest conditions entailed by $P(x)$ and $\neg P(x)$) may not be computable or are too expensive to compute. For linear arithmetics, however, we can use off-the-shelf automated theorem provers to obtain such conditions.

In rule A-APP, each argument v_i is abstracted by using the abstraction type σ_i with $y_1, \ldots, y_{i-1}$ being replaced by the actual arguments. Note that this rule applies also to the case where the sequence $\widetilde{v}$ is empty (i.e. $k = 0$). Thus, we can derive $\Gamma \vdash y : \sigma \leadsto y$ if $\Gamma(y) = \sigma$. Note also that the boolean expression e_i in A-APP can depend on $y_1, \ldots, y_{i-1}$.

In A-FAIL, the assume statement is inserted for filtering out an invalid combination of abstract values. For example, let Γ be

$x : \mathtt{int}[\lambda x.x > 0, \lambda x.x < 0]$. Then, $\mathtt{assume}\ (\sharp_1(x) \wedge \sharp_2(x))$; is inserted since $x > 0$ and $x < 0$ cannot be true simultaneously. In A-ASM, we can use the fact that v is $\mathtt{true}$ in e for abstracting e.

Rule A-CFUN is used for changing the abstraction type of a function from $x : \sigma_1 \to \sigma_2$ to $x : \sigma_1' \to \sigma_2'$, which is analogous to the usual rule for subtyping-based coercion. If a function f is used in different contexts which require different abstraction types of f, A-CFUN can be used to adjust the abstraction type of f to that required for each context.

We can read the predicate abstraction rules for $\Gamma \vdash e : \sigma \rightsquigarrow e'$ as an algorithm that takes Γ, e and σ as input, and outputs e' as an abstraction of e, by (1) restricting applications of the rules for coercion (of names A-CXYZ) to the actual arguments of function applications, and (2) fixing an algorithm to find the boolean formulas ψ and ψ' in A-CADD. (Note that in A-LET, the type σ' can be obtained from Γ and e_1.) The rule for $\vdash D : \Gamma \rightsquigarrow D'$ can then be interpreted as an algorithm that takes D and Γ as input, and outputs an HBP D' as an abstraction of D.

Example 4.1. Recall the program M_2 in Section 1. Let Γ be:

$$x : \mathtt{int}[\lambda\nu.\nu \geq 0], g : \mathtt{int}[\lambda\nu.\nu > 0] \to \star$$

The body of f is transformed as follows. $x + 1$ is transformed by:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x+1 : \mathtt{int}[\lambda\nu.\nu = x+1] \rightsquigarrow \mathtt{true}}\ \text{A-BASE}}{\Gamma \vdash x+1 : \mathtt{int}[\lambda\nu.\nu = x+1, \lambda\nu.\nu > 0] \rightsquigarrow e_1}\ \text{A-CADD}}{\Gamma \vdash x+1 : \mathtt{int}[\lambda\nu.\nu > 0] \rightsquigarrow e_2}\ \text{A-CREM}$$

Here, $e_1 \equiv \mathtt{let}\ y_1 = \mathtt{true}\ \mathtt{in}\ \langle y_1, e_3\rangle$ and $e_2 \equiv \mathtt{let}\ \langle y_1, y_2\rangle = e_1\ \mathtt{in}\ y_2$, with $e_3 \equiv (\mathtt{assume}\ \mathtt{true}; \mathtt{true}) \blacksquare (\mathtt{assume}\ \neg(x \wedge y_1); \mathtt{false})$. Here, we used $\mathtt{true}$ and $\neg(x \wedge y_1)$ as ψ and ψ' respectively, in A-CADD. (Note that $P(y_1) \Rightarrow \theta_{\Gamma, y_1:\mathtt{int}[\lambda\nu.\nu=x+1]}(\psi_1)$, i.e., $y_1 \leq 0 \Rightarrow \neg(x \geq 0 \wedge y_1 = x+1)$ holds.) By simplifying e_2, we get $\mathtt{if}\ x\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ \mathtt{true}\blacksquare\mathtt{false}$. Thus, the body $g(x+1)$ of function f is transformed by using A-APP as follows:

$$\dfrac{\Gamma(g) = \mathtt{int}[\lambda\nu.\nu > 0] \to \star \qquad \dfrac{\vdots}{\Gamma \vdash x+1 : \mathtt{int}[\lambda\nu.\nu > 0] \rightsquigarrow e_2}}{\Gamma \vdash g(x+1) : \star \rightsquigarrow \begin{array}{l}\mathtt{let}\ y = \mathtt{if}\ x\ \mathtt{then}\ \mathtt{true} \\ \qquad \mathtt{else}\ (\mathtt{true}\blacksquare\mathtt{false})\ \mathtt{in}\ g(y)\end{array}}$$

Our predicate abstraction rules are applicable to programs that use infinite data domain other than integers. See [25] for an example of abstracting a list-processing program.

We discuss properties of the predicate abstraction relation below. First, we show that if abstraction types are consistent, there is always a valid transformation. We write $\Gamma \vdash_{\mathtt{AT}} e : \sigma$ for the type judgment relation obtained from the predicate abstraction rules by removing all the conditions on outputs: see [25].

Theorem 4.1. *Suppose* $\Gamma \vdash_{\mathtt{AT}} e : \sigma$. *Then,* $\mathtt{A2S}(\Gamma) \vdash_{\mathtt{ST}} e : \mathtt{A2S}(\sigma)$. *Furthermore, there exists* e' *such that* $\Gamma \vdash e : \sigma \rightsquigarrow e'$.

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathtt{AT}} e : \sigma$. Note that in the rule for A-CADD, we can choose $\mathtt{true}$ as ψ and ψ'. □

The following lemma guarantees that the output of the transformation is well-typed.

Lemma 4.2. *If* $\Gamma \vdash e_1 : \sigma \rightsquigarrow e_2$, *then* $\beta(\Gamma) \vdash_{\mathtt{ST}} e_2 : \beta(\sigma)$.

Proof. Straightforward induction on the derivation of $\Gamma \vdash e_1 : \sigma \rightsquigarrow e_2$. □

The theorem below states that our predicate abstraction is sound in the sense that if a source program fails, so does its abstraction (see [25] for the proof). Thus, the safety of the abstract program (which is decidable by Theorem 3.1) is a sufficient condition for the safety of the source program.

Theorem 4.3 (soundness). *If* $\vdash D_1 : \Gamma \rightsquigarrow D_2$ *and* $main\langle\rangle \stackrel{s}{\Longrightarrow}_{D_1} \mathtt{fail}$, *then* $main\langle\rangle \stackrel{s}{\Longrightarrow}_{D_2} \mathtt{fail}$.

The theorem above says that the abstraction is sound but not how good the abstraction is. We compare below the verification power of the combination of predicate abstraction and higher-order model checking with the dependent intersection type system given in [25], which is essentially equivalent to the one in [32].

We write $\mathcal{B}[\psi_1, \ldots, \psi_k]$ for the set of formulas constructed from $\psi_1, \ldots, \psi_k$ and boolean operators ($\mathtt{true}, \mathtt{false}, \wedge, \vee, \neg$). For an abstraction type σ, the set *DepTy*(σ) of dependent types is:

$$\begin{array}{l} \mathit{DepTy}(b[P_1, \ldots, P_n]) = \{\{\nu : b \mid \psi\} \mid \psi \in \mathcal{B}[P_1(\nu), \ldots, P_n(\nu)]\} \\ \mathit{DepTy}(x : \sigma_1 \to \sigma_2) = \{(x : \delta_{11} \to \delta_{21}) \wedge \cdots \wedge (x : \delta_{1m} \to \delta_{2m}) \\ \qquad \mid \delta_{11}, \ldots, \delta_{1m} \in \mathit{DepTy}(\sigma_1), \delta_{21}, \ldots, \delta_{2m} \in \mathit{DepTy}(\sigma_2)\} \end{array}$$

We extend *DepTy* to a map from abstraction type environments to the powerset of dependent type environments by:

$$\begin{array}{l} \mathit{DepTy}(\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}) = \\ \{\{x_1 : \delta_1, \ldots, x_n : \delta_n\} \mid \delta_i \in \mathit{DepTy}(\sigma_i) \text{ for each } i \in \{1, \ldots, n\}\} \end{array}$$

The following theorem says that our predicate abstraction (with higher-order model checking) has at least the same verification power as the dependent intersection type system.

Theorem 4.4 (relative completeness). *Suppose* $\vdash_{\mathit{DIT}} D : \Delta$. *If* $\Delta \in \mathit{DepTy}(\Gamma)$, *then there exists* D' *such that* $\vdash D : \Gamma \rightsquigarrow D'$ *and* $main\langle\rangle \not\Longrightarrow_{D'} \mathtt{fail}$.

Remark 1. The converse of the above theorem does not hold: see [25]. Together with Theorem 4.4, this implies that our combination of predicate abstraction and higher-order model checking is strictly more powerful than the dependent intersection type system.

Remark 2. The well-formedness condition for abstraction types is sometimes too restrictive to express a necessary predicate. For example, consider the following program.

```
let apply f x = f x in let g y z = assert(y=z) in
let k n = apply (g n) n; k(n+1) in  k(0)
```

In order to verify that the assertion failure does not occur, we need a correlation between the argument of f and x, which cannot be expressed by abstraction types. The problem can be avoided either by adding a dummy parameter to `apply` (as `let apply n f x = ...`) and using the abstraction type $n : \mathtt{int}[\,] \to (\mathtt{int}[\lambda\nu.\nu = n] \to \star) \to \mathtt{int}[\lambda\nu.\nu = n] \to \star$, or by swapping the parameters f and x. A more fundamental solution (which is left for future work) would be to introduce *polymorphic* abstraction types, like $\forall m : \mathtt{int}.(\mathtt{int}[\lambda\nu.\nu = m] \to \star) \to \mathtt{int}[\lambda\nu.\nu = m] \to \star$, and extend the predicate abstraction rules accordingly.

5. Counterexample-Guided Abstraction Refinement (CEGAR)

This section describes a CEGAR procedure to discover new predicates used for predicate abstraction when the higher-order model checker TRECS has reported an error path s of a boolean program.

5.1 Feasibility checking

Given an error path s of an abstract program, we first check whether s is feasible in the source program D, i.e. whether $main\langle\rangle \stackrel{s}{\Longrightarrow}_D \mathtt{fail}$. This can be easily checked by actually executing the source program along the path s, and checking whether all the branching conditions are true. (Here, we assume that the program is closed. If we allow free variables for storing base values, we can just

symbolically execute the source program along the path, and check whether all the conditions are satisfiable.) If the source program indeed has the error path (i.e. $\mathit{main}\langle\rangle \overset{s}{\Longrightarrow}_D \texttt{fail}$), then we report the error path as a counterexample.

5.2 Predicate discovery and refinement of abstraction types

If the error path is infeasible (i.e. $\mathit{main}\langle\rangle \overset{s}{\not\Longrightarrow}_D \texttt{fail}$), we find new predicates to refine predicate abstractions.

In the case of the model checking of first-order programs, this is usually performed by, for each program point ℓ in the error path, (i) computing the strongest condition C_1 at ℓ, (ii) computing the weakest condition C_2 for reaching from ℓ to the failure node, and (iii) using a theorem prover to find a condition C such that $C_1 \Rightarrow C$ and $C \Rightarrow \neg C_2$. Then the predicates in C can be used for abstracting the state at the program point. For example, in the reduction sequence (2) of M_1 in Section 1, the condition C_1 on the local variable x is $n > 0 \wedge x = n$, and the condition C_2 is $x + 1 \geq 0$. From them, we obtain $C \equiv x > 0$ as a predicate for abstracting x.

It is unclear, however, how to extend it to deal with higher-order functions. For example, in the example above, how can we find a suitable abstraction type for functions f and g? To address this issue, as mentioned in Section 1, we use the following type-based approach. From an infeasible error path, we first construct a *straightline higher-order program* (abbreviated to SHP, which is straightline in the sense that it contains neither branches nor recursion and that each function is called at most once) that has exactly one execution path, corresponding to the path s of the source program. We then infer the dependent types of functions in the straightline program, and use the predicates occurring in the dependent types for refining abstraction types of the source program. We describe each step in more detail below.

5.2.1 Constructing SHP

Given a source program and a path s, the corresponding SHP is obtained by (i) making a copy of each function for each call in the execution path, and (ii) for each copy, removing the branches not taken in s.

Example 5.1. Recall the program M_3 in Section 1.

$$\begin{array}{l}
\mathit{main}\langle\rangle = k\ m \qquad f\ x\ g = g(x+1) \\
h\ z\ y = (\texttt{assume}\ y > z; \langle\rangle) \,\Box\, (\texttt{assume}\ \neg(y > z); \texttt{fail}) \\
k\ n = (\texttt{assume}\ n \geq 0; f\ n\ (h\ n)) \Box (\texttt{assume}\ \neg(n \geq 0); \langle\rangle)
\end{array}$$

Here, we have represented conditionals and assert expressions in our language.[7] Given the spurious error path $0 \cdot 1$, we obtain the following SHP.

$$\begin{array}{ll}
\mathit{main}\langle\rangle = k\ m & h\ z\ y = \texttt{assume}\ \neg(y > z); \texttt{fail} \\
f\ x\ g = g(x+1) & k\ n = \texttt{assume}\ n \geq 0; f\ n\ (h\ n)
\end{array}$$

It has been obtained by removing irrelevant non-deterministic branches in h and k.

The construction of an SHP generally requires duplication of function definitions and function parameters. For example, consider the following program:

$$\begin{array}{l}
\mathit{main}\langle\rangle = k\ m \qquad \mathit{twice}\ f\ x = f(f\ x) \\
g\ x = \texttt{if}\ x \leq 0\ \texttt{then}\ 1\ \texttt{else}\ 2 + g(x-1) \\
k\ n = \texttt{let}\ x = \mathit{twice}\ g\ n\ \texttt{in}\ \texttt{assert}\ (x > 0)
\end{array}$$

(where m is some integer constant). The program calls the function g twice, and asserts that the result x is positive. Suppose that an infeasible path 0101 has been given, which represents the following (infeasible) execution path:

$$\begin{array}{l}
\mathit{main}\langle\rangle \longrightarrow k\ m \Longrightarrow \texttt{let}\ x = g(g\ m)\ \texttt{in} \cdots \\
\quad \overset{0}{\Longrightarrow} \texttt{let}\ x = g(1)\ \texttt{in} \cdots \overset{1}{\Longrightarrow} \texttt{let}\ x = 2 + g(0)\ \texttt{in} \cdots \\
\quad \overset{0}{\Longrightarrow} \texttt{let}\ x = 2 + 1\ \texttt{in} \cdots \Longrightarrow \texttt{assert}\ 3 > 0 \overset{1}{\Longrightarrow} \texttt{fail}
\end{array}$$

The path is infeasible because the final transition is invalid.

From the source program and the path above, we construct the following straightline program:[8]

$$\begin{array}{l}
\mathit{main}\langle\rangle = k\ m \qquad \mathit{twice}\ \langle f^{(1)}, f^{(2)}\rangle\ x = f^{(2)}(f^{(1)}\ x) \\
g^{(1)}\ x = \texttt{assume}\ x \leq 0; 1 \qquad g^{(3)}\ x = \texttt{assume}\ x \leq 0; 1 \\
g^{(2)}\ x = \texttt{assume}\ \neg(x \leq 0); 2 + g^{(3)}(x-1) \\
k\ n = \texttt{let}\ x = \mathit{twice}\ \langle g^{(1)}, g^{(2)}\rangle\ n\ \texttt{in}\ \texttt{assume}\ \neg(x > 0); \texttt{fail}
\end{array}$$

As g is called three times, we have prepared three copies $g^{(1)}$, $g^{(2)}$, $g^{(3)}$ of g, and eliminated unused non-deterministic branches. Note that the function parameter f of *twice* has been replaced by a function pair $\langle f^{(1)}, f^{(2)}\rangle$ accordingly.

The general construction is given below. Consider a program normalized to the following form:

$$\begin{array}{rcl}
D & ::= & \{f_1\ \widetilde{x}_1 = e_{10} \Box e_{11}, \ldots, f_m\ \widetilde{x}_m = e_{m0} \Box e_{m1}\} \\
e & ::= & \texttt{assume}\ v; a \mid \texttt{let}\ x = \mathrm{op}(\widetilde{v})\ \texttt{in}\ a \\
a & ::= & \langle\rangle \mid x\ \widetilde{v} \mid f\ \widetilde{v} \mid \texttt{fail} \\
v & ::= & c \mid x\ \widetilde{v} \mid f\ \widetilde{v}
\end{array}$$

Here, for the sake of simplicity, we have assumed that every function definition has at most one (tail) function call, and the return value is $\langle\rangle$; this does not lose generality as the normal form can be obtained by applying CPS transformation and λ-lifting. Given a path $s = b_1 \cdots b_\ell$ of D (which means that the branch b_i has been chosen at ith function call), the corresponding SHP $D' = \mathbf{SHP}(D, s)$ is given by:

$$\begin{array}{l}
D' = \{f_i^{(j)}\ \widetilde{x}_i = [e_{ib_j}]_{j+1} \mid i \in \{1, \ldots, m\}, j \in \{1, \ldots, \ell\}, \\
\qquad \text{the target of the } j\text{th function call is } f_i\} \\
\quad \cup \{f_i^{(j)}\ \widetilde{x}_i = \langle\rangle \mid i \in \{1, \ldots, m\}, j \in \{1, \ldots, \ell\}, \\
\qquad \text{the target of the } j\text{th function call is not } f_i\} \\
\quad \cup \{\mathit{main}\langle\rangle = \mathit{main}^{(1)}\langle\rangle\}
\end{array}$$

Here, $[e]_j$ is given by:

$$\begin{array}{l}
[\texttt{assume}\ v; a]_j = \texttt{assume}\ v; [a]_j \\
[\texttt{let}\ x = \mathrm{op}(\widetilde{v})\ \texttt{in}\ a]_j = \texttt{let}\ x = \mathrm{op}(\widetilde{v})\ \texttt{in}\ [a]_j \\
[\langle\rangle]_j = \langle\rangle \qquad [\texttt{fail}]_j = \texttt{fail} \qquad [x]_j = x \\
[x\ v_1\ \cdots\ v_k]_j = \sharp_j(x)\ {v_1}^{\flat_{j+1}}\ \cdots\ {v_k}^{\flat_{j+1}} \qquad (k \geq 1) \\
[f\ v_1\ \cdots\ v_k]_j = f^{(j)}\ {v_1}^{\flat_{j+1}}\ \cdots\ {v_k}^{\flat_{j+1}} \\
c^{\flat_j} = c \qquad x^{\flat_j} = x \quad (\text{if } x \text{ is a base variable}) \\
(x\ \widetilde{v})^{\flat_j} = \langle \underbrace{\lambda\widetilde{y}.\langle\rangle, \ldots, \lambda\widetilde{y}.\langle\rangle}_{j-1}, \sharp_j(x)(\widetilde{v}^{\flat_j}), \ldots, \sharp_\ell(x)(\widetilde{v}^{\flat_j})\rangle \\
\quad (\text{if } x \text{ is a function variable}) \\
(f\ \widetilde{v})^{\flat_j} = \langle \underbrace{\lambda\widetilde{y}.\langle\rangle, \ldots, \lambda\widetilde{y}.\langle\rangle}_{j-1}, f^{(j)}(\widetilde{v}^{\flat_j}), \ldots, f^{(\ell)}(\widetilde{v}^{\flat_j})\rangle
\end{array}$$

Here, each function parameter has been replaced by a ℓ-tuple of functions.

The SHP $\mathbf{SHP}(D, s)$, constructed from a source program D and a spurious error path s, contains neither recursion nor non-deterministic branch, and is reduced to $\texttt{fail}$ if and only if $\mathit{main}\langle\rangle \overset{s}{\Longrightarrow}_D \texttt{fail}$. Furthermore, each function in the SHP is called at most once.

[7] Here, for the sake of simplicity, we assume that m is some integer constant. As already mentioned, the random number generator *randi* can actually be encoded in our language.

[8] For clarity, we have extended our language with tuples of functions. If necessary, they can be removed by the currying transformation.

The generated straightline program satisfies the following properties.

Lemma 5.1. *Suppose $D' = \textbf{SHP}(D, s)$. Then:*

1. *D' contains neither recursions nor non-deterministic branches $e_1 \Box e_2$.*
2. *$main\langle\rangle \stackrel{s}{\Longrightarrow}_D$ `fail` if and only if $main\langle\rangle \Longrightarrow_{D'}$ `fail`.*
3. *Each function $f_i^{(j)}$ in D' is called at most once.*

5.2.2 Typing SHP

The next step is to infer dependent types for functions in SHP. Thanks to the properties that SHP contains neither recursion nor non-deterministic branch and that every function is linear, the standard dependent type system is sound and complete for the safety of the program. Let us write $\vdash_{\mathtt{DT}} D$ if D is typable in the fragment of the dependent type system presented in Section 4 without intersection types (but extended with (non-dependent) tuple types).

Lemma 5.2. *Let $D' = \textbf{SHP}(D, s)$. Then, $\vdash_{DT} D' : \Delta$ for some Δ if and only if $main\langle\rangle \not\Longrightarrow_{D'}$ `fail`.*

Proof sketch The "only if" part follows immediately from the soundness of the dependent type system. For the "if" part, it suffices to observe that, as every function in D' is linear, each variable x of base type can be assigned a type $\{\nu : b \mid \nu = v\}$, where v is the value that x is bound to. □

We can use existing algorithms [32, 33] to infer dependent types: we first prepare a template of a dependent type for each function, generate constraints on predicate variables, and solve the constraints. We give below an overview of the dependent type inference procedure through an example; an interested reader may wish to consult [32, 33].

Example 5.2. Recall the straightline program in Example 5.1. We prepare the following templates of the types of functions f, h, k:

$$\begin{array}{l} f : (x : \{\nu : \mathtt{int} \mid P_1(\nu)\} \to (y : \{\nu : \mathtt{int} \mid P_2(\nu, x)\} \to \star) \to \star) \\ h : (z : \{\nu : \mathtt{int} \mid P_3(\nu)\} \to y : \{\nu : \mathtt{int} \mid P_4(\nu, z)\} \to \star) \\ k : (x : \{\nu : \mathtt{int} \mid P_0(\nu)\} \to \star) \end{array}$$

From the program, we obtain the following constraints on $P_0, \ldots, P_4$:

$$\begin{array}{l} P_0(m) \qquad \forall x.(P_1(x) \Rightarrow P_2(x+1, x)) \\ \forall z, y.(P_3(z) \wedge P_4(y, z) \Rightarrow y > z) \\ \forall n, y.P_0(n) \Rightarrow \\ \qquad (n \geq 0 \Rightarrow (P_1(n) \wedge P_3(n) \wedge (P_2(y, n) \Rightarrow P_4(y, n)))) \end{array}$$

Each constraint has been obtained from the definitions of *main*, f, g, and k. They can be normalized to:

$$\begin{array}{l} \forall \nu.(\nu = m \Rightarrow P_0(\nu)) \\ \forall n, \nu.(P_0(n) \wedge n \geq 0 \wedge \nu = n \Rightarrow P_1(\nu)) \\ \forall x, \nu.(P_1(x) \wedge \nu = x + 1 \Rightarrow P_2(\nu, x)) \\ \forall n, \nu.(P_0(n) \wedge n \geq 0 \wedge \nu = n \Rightarrow P_3(\nu)) \\ \forall n, z, \nu.(P_0(n) \wedge n \geq 0 \wedge z = n \wedge P_2(\nu, n) \Rightarrow P_4(\nu, z)) \\ \forall z, y.(P_3(z) \wedge P_4(y, z) \Rightarrow y > z) \end{array}$$

These constraints are "acyclic" in the sense that for each constraint of the form $C_i \Rightarrow P_i(\widetilde{x})$, C_i contains only (positive) occurrences of predicates P_j's such that $j < i$ occur. Such constraints can be solved by using a sub-procedure of existing methods for dependent type inference based on interpolants [32, 33], and the following predicates can be obtained. (The inferred predicates depend on the underlying interpolating theorem prover.)

$$\begin{array}{l} P_0(\nu) \equiv P_1(\nu) \equiv P_3(\nu) \equiv \mathtt{true} \\ P_2(\nu, x) \equiv P_4(\nu, x) \equiv \nu > x \end{array}$$

Thus, we obtain the following types for f and h:

$$\begin{array}{l} f : (x : \{\nu : \mathtt{int} \mid \mathtt{true}\} \to (y : \{\nu : \mathtt{int} \mid \nu > x\} \to \star) \to \star) \\ h : (z : \{\nu : \mathtt{int} \mid \mathtt{true}\} \to y : \{\nu : \mathtt{int} \mid \nu > z\} \to \star) \end{array}$$

5.2.3 Refining abstraction types

The final step is to refine the abstraction types of the source program, based on the dependent types inferred for the straightline program. Let $\delta_{f,j}$ be the inferred dependent type of $f^{(j)}$. Then, we can obtain an abstraction type $\sigma_{f,j}$ such that $\mathtt{undup}(\delta_{f,j}) \in \mathit{DepTy}(\sigma_{f,j})$ (the choice of such $\sigma_{f,j}$ depends on what predicates are considered atomic), where $\mathtt{undup}(\delta)$ is defined by:

$$\begin{array}{rcl} \mathtt{undup}(\{\nu : b \mid \psi\}) & = & \{\nu : b \mid \psi\} \\ \mathtt{undup}(x : \delta_1 \to \delta_2) & = & x : \mathtt{undup}(\delta_1) \to \mathtt{undup}(\delta_2) \\ \mathtt{undup}(\delta_1 \times \cdots \times \delta_n) & = & \bigwedge_{i \in \{1,\ldots,n\}} \mathtt{undup}(\delta_i) \end{array}$$

The new abstraction type σ'_f of f is given by:

$$\sigma'_f = \sigma_f \sqcup \sigma_{f,1} \sqcup \cdots \sqcup \sigma_{f,\ell},$$

where σ_f is the previous abstraction type of f and $\sigma_1 \sqcup \sigma_2$ is obtained by just merging the corresponding predicates:

$$\begin{array}{l} b[\widetilde{P}] \sqcup b[\widetilde{Q}] = b[\widetilde{P}, \widetilde{Q}] \\ (x : \sigma_1 \to \sigma_2) \sqcup (x : \sigma'_1 \to \sigma'_2) = x : (\sigma_1 \sqcup \sigma'_1) \to (\sigma_2 \sqcup \sigma'_2) \end{array}$$

We write *Refine*(Γ, Δ) for the refined abstraction type environment $f_1 : \sigma'_{f_1}, \ldots, f_n : \sigma'_{f_n}$. (There is a non-determinism coming from the choice of $\sigma_{f,j}$, but that does not matter below.)

Example 5.3. Recall Example 5.3. From the dependent types of f and g, we obtain the following abstraction types:

$$\begin{array}{l} f : (x : \mathtt{int}[\,] \to (y : \mathtt{int}[\lambda\nu.\nu > x] \to \star) \to \star) \\ h : (z : \mathtt{int}[\,] \to y : \mathtt{int}[\lambda\nu.\nu > z] \to \star) \end{array}$$

Suppose that the previous abstraction types were

$$\begin{array}{l} f : (x : \mathtt{int}[\,] \to (y : \mathtt{int}[\,] \to \star) \to \star) \\ h : (z : \mathtt{int}[\lambda\nu.\nu = 0] \to y : \mathtt{int}[\lambda\nu.\nu > 0] \to \star) \end{array}$$

Then, the refined abstraction types are:

$$\begin{array}{l} f : (x : \mathtt{int}[\,] \to (y : \mathtt{int}[\lambda\nu.\nu > x] \to \star) \to \star) \\ h : (z : \mathtt{int}[\lambda\nu.\nu = 0] \to y : \mathtt{int}[\lambda\nu.\nu > 0, \lambda\nu.\nu > z] \to \star) \end{array}$$

5.3 Properties of the CEGAR algorithm

We now discuss properties of the overall CEGAR algorithm. If the refined abstraction type is obtained from an infeasible error path s, the new abstract boolean program no longer has the path s. This is the so called "progress property" known in the literature on CEGAR for the usual (i.e. finite state or pushdown) model checking. Formally, we can prove the following property (see [25] for the proof):

Theorem 5.3 (progress). *Let D_1 be a well-typed program and s be an infeasible path of D_1. Suppose $D_2 = \textbf{SHP}(D_1, s)$ and $\vdash_{DIT} D_2 : \Delta$ with $\Gamma = \mathit{Refine}(\Gamma', \Delta)$ for some Γ'. Then, there exists D_3 such that $\vdash D_1 : \Gamma \rightsquigarrow D_3$, and $main\langle\rangle \stackrel{s}{\not\Longrightarrow}_{D_3}$ `fail`.*

The progress property above does not guarantee that the verification will eventually terminate: There is a case where the entire CEGAR loop does not terminate, finding new spurious error paths forever (see Section 6). Indeed, we cannot expect to obtain a sound and complete verification algorithm, as the reachability is undecidable in general even if programs are restricted to those using only linear arithmetics.

We can however modify our algorithm so that it is *relatively complete* with respect to the dependent intersection type system, in the sense that all the programs typable in the dependent intersection type system can be verified by our method. Let *genP* be a total map from the set of integers to the set of predicates. (Such a total map exists, as the set of predicates is recursively enumerable.) Upon the i-th iteration of the CEGAR loop, add the predicate *genP*(i)

to each position of abstraction type, in addition to the predicates inferred from counterexamples. Then, if a program is well-typed under Δ in the dependent intersection type system, an abstraction type environment Γ such that $\Delta \in \mathit{DepTy}(\Gamma)$ is eventually found, so that by Theorem 4.4, our verification succeeds. Of course, this is impractical, but we may be able to adapt the technique of [18] to get a practical algorithm.

6. Implementation and Preliminary Experiments

Based on our method described so far, we have implemented a prototype verifier for a tiny subset of Objective Caml, having only booleans and integers as base types. Instead of the non-deterministic choice ($e_1 \Box e_2$), the system allows conditionals and free variables (representing unknown integers). Our verifier uses TRECS [21, 22] as the underlying higher-order model checker (for Step 2 in Figure 1), and uses CSIsat [6] for computing interpolants to solve constraints (for Step 4). CVC3 [5] is used for feasibility checking (for Step 3) and computing abstract transitions (i.e., to compute formulas ψ and ψ' in rule A-CADD of Figure 4 for Step 1). As computing the precise abstract transitions (i.e. the strongest formulas ψ and ψ' in rule A-CADD) is expensive, we have adapted several optimizations described in Section 5.2 of [2] such as bounding the maximum number of predicates taken into account for computing abstraction with a sacrifice of the precision. The implementation can be tested at `http://www.kb.ecei.tohoku.ac.jp/~ryosuke/cegar/`. The full version [25] contains more details about the experiments.

The results of preliminary experiments are shown in Table 1. The column "S" shows the size of programs, measured in word counts. The column "O" shows the largest order of functions in the program (an order-1 function takes only base values as arguments, while an order-2 function takes order-1 functions as arguments).[9] The column "C" shows the number of CEGAR cycles. The remaining columns show running times, measured in seconds. The column "abst" shows the time spent for computing abstract programs (from given programs and abstraction types). The column "mc" shows the time spent (by TRECS) for higher-order model checking. The column "cegar" shows the time spent for finding new predicates (Step 4 in Figure 1). The column "total" shows the total running time (machine spec.: 3GHz CPU with 8GB memory).

The programs used in the experiment are as follows. Free variables denote unknown integers.

- `intro1`, `intro2`, and `intro3` are the three examples in Section 1.
- `sum` and `mult` compute $1 + \cdots + n$ and $\underbrace{n + \cdots + n}_{n}$ respectively, and asserts that the result is greater than or equal to n. Here is the code of `sum`.

```
let rec sum n =
  if n <= 0 then 0 else n + sum (n - 1)
in assert (n <= sum n)
```

- `max` defines a higher-order function that takes a function that computes the maximum of two integers, and three integers as input, and returns the maximum of the three integers:

```
let max max2 x y z = max2 (max2 x y) z in
let f x y = if x >= y then x else y in
let m = max f x y z in assert (f x m = m)
```

The last line asserts that the return value of `max` is greater than or equal to `x` (with respect to the function f).

[9] Because of the restriction of the model checker TRECS, all the source programs are actually verified after the CPS transformation. Thus, all the tested programs are actually higher-order, taking continuation functions.

- `mc91` is McCarthy 91 function.

```
let rec mc91 x =
  if x > 100 then x - 10 else mc91(mc91(x + 11))
in if n <= 101 then assert (mc91 n = 91)
```

The last line asserts that the result is 91 if the argument is less than or equal to 101.

- `ack` defines Ackermann function *ack* and asserts $\mathit{ack}(n) \geq n$.
- `repeat` defines a higher-order function that takes a function f and integers n, s, then returns $f^n(s)$.

```
let rec repeat f n s =
  if n = 0 then s else f (repeat f (n - 1) s) in
let succ x = x + 1 in
  assert (repeat succ n 0 = n)
```

- `fhnhn` is a program not typable in the dependent intersection type system but verifiable in our method (c.f. Remark 1):

```
let f x y = assert (not (x() > 0 && y() < 0)) in
let h x y = x in let g n = f (h n) (h n) in g m
```

- `hrec` is a program that creates infinitely many function closures:

```
let rec f g x =
  if x >= 0 then g x else f (f g) (g x) in
let succ x = x + 1 in assert(f succ n >= 0)
```

- `neg` is an example that needs nested intersection types:

```
let g x y = x in
let twice f x y = f (f x) y in
let neg x y = -x() in
  if n >= 0 then assert(twice neg (g n) () >= 0)
  else ()
```

- `apply` is the program discussed in Remark 2.
- `a-prod`, `a-cppr`, and `a-init` are programs manipulating arrays. A (functional) array has been encoded as a pair of the size and a function from indices to array contents. For example, the functions for creating and updating arrays are defined as follows.

```
let mk_array n i = assert(0<=i && i<n); 0
let update i n a x =
  a(i); let a' j = if i=j then x else a(i) in a'
```

For `a-prod` and `a-cppr`, it has been verified that there is no array boundary error. Program `a-init` initializes an array, and asserts the correctness of initialization. and `a-max` creates an array of size n whose i-th element is $n-i$, computes the maximum element m, and asserts that $m \geq n$. These examples show an advantage of higher-order model checking; various data structures can be encoded as higher-order functions, and their properties can be verified in a uniform manner.

- `l-zipunzip` and `l-zipmap` are taken from list-processing programs. We have manually abstracted lists to integers (representing the list length), and then verified the size properties of list functions. For example, the code for l-zipunizp is:

```
let f g x y = g (x+1) (y+1) in
let rec unzip x k =
  if x=0 then k 0 0 else unzip (x-1) (f k) in
let rec zip x y =
  if x=0 then if y=0 then 0 else fail()
  else if y=0 then fail() else 1+zip(x-1)(y-1)
in unzip n zip
```

- `hors` encodes a model checking problem for higher-order recursion schemes *extended with integers* (which cannot be handled by recursion scheme model checkers).
- `e-simpl` and `e-fact` model programs that use exceptions, where an exception handler is expressed as a continuation, and assert that there are no uncaught exceptions. The idea of the encoding of exceptions is similar to [20], but unlike [20], exceptions can carry integer values.
- `r-lock` and `r-file` model programs that use locks and files, and assert that they are accessed in a correct manner. The encoding

is similar to [22], but (unlike [22]) the programs' control behaviors depend on integer values.

- A program of name "xxx-e" is a buggy version of the program "xxx".

The above programs have been verified (or rejected, for wrong programs) correctly, except `apply`. As discussed in Remark 2, `apply` cannot be verified because of the fundamental limitation of abstraction types. Our system continues to infer new (but too specific) abstraction types $(\texttt{int}[\lambda\nu.\nu = i] \rightarrow \star) \rightarrow \texttt{int}[\lambda\nu.\nu = i] \rightarrow \star$ for $i = 0, 1, 2, \ldots$ forever and (necessarily) does not terminate. The program can however be verified if the arguments of `apply` are swapped. The same problem has been observed for variations of some of the programs above: sometimes we had to add or swap arguments of functions.

Another limitation revealed by the experiments is that for some variations of the programs, the system infers too specific predicates and does not terminate. For example, the verification for `a-max` fails if we assert $m \geq a(j)$ instead of $m \geq n$ (where m is the maximal element computed, a is the array, and j is some index). Relaxing these limitations seems necessary for verification of larger programs, and we plan to do so by adding heuristics to generalize inferred abstraction types (e.g. by using widening techniques [9]).

Apart from the limitations above, our system is reasonably fast. This indicates that, although higher-order model checking has the extremely high worst-case complexity (n-EXPTIME complete [28]), our overall approach works at least for small programs as long as suitable predicates are found. See further discussions on the scalability in Section 8.

7. Related Work

7.1 Model Checking of Higher-Order Programs

The model checking of higher-order recursion schemes has been extensively studied [19, 24, 28]. Ong [28] proved the decidability of the modal μ-calculus model checking of recursion schemes. Kobayashi [22] then proposed a new framework of higher-order program verification based on the model checking of recursion schemes, already suggesting a use of predicate abstraction and CEGAR to deal with programs manipulating infinite data domain. There were two missing pieces in his framework, however. One was a practical model checking algorithm for recursion schemes (note that the model checking of recursion schemes is in general n-EXPTIME-complete), and the other was a method to apply predicate abstraction and CEGAR to higher-order programs. The former piece has been supplied later by Kobayashi [20], and supplying the latter piece was the goal of the present paper.

In parallel to the present work, Unno et al. [26, 34] and Ong and Ramsay [29] proposed applications of higher-order model checking to verification of tree-processing programs. Their approaches are radically different from ours. First, they use different abstraction techniques: tree data are abstracted using either tree automata [26, 34] or patterns [29], which cannot abstract values using binary predicates (such as $2 \times x \geq y$). Secondly, The method of [26] applies only to programs that can be expressed in the form of (higher-order) tree transducers, and the extension in [34] requires user annotations. Ong and Ramsay's method [29] applies to general functional programs and includes a CEGAR mechanism, but the precision of their method is heavily affected by that of a variable binding analysis, and their CEGAR is completely different from ours. Their technique does not satisfy relative completeness like Theorem 4.4.

7.2 Dependent Type Inference

There have been studies on automatic or semi-automatic inference of dependent types [7, 11, 16, 31–33]. There are similarities between the goals of those studies and that of our work. First, one of the goals of dependent type inference is to prove the lack of assertion failures, as in the present work. Secondly, our technique can actually be used for inferring dependent types. Recursion scheme model checker TRECS [20] is type-based, and produces type information as a certificate of successful verification. For example, for the abstraction of the last example in Section 1, it infers the type $\star \rightarrow (\texttt{true} \rightarrow \star) \rightarrow \star$ for (the abstract version of) f. Combined with the abstraction type of f, we can recover the following dependent type for $f : (x : \texttt{int} \rightarrow (y : \{\nu : \texttt{int} \mid \nu > x\} \rightarrow \star) \rightarrow \star)$.

Though the goals are similar, the techniques are different. Rondon et al.'s liquid types [31] requires users to specify predicates (or more precisely, shapes of predicates, called *qualifiers*) used in dependent types. Jhala et al. [16] proposed an automatic method for inferring qualifiers for liquid types. Their method extracts qualifiers from a proof that a finite unfolding of a source program does not get stuck, and has some similarity to our method to infer abstraction types from an error path. Unno and Kobayashi [33] proposed an automatic method for inferring dependent types. They first prepare templates of dependent types (that contain predicate variables) and generate (possibly recursive) constraints on predicate variables. They then solve the constraints by using an interpolating theorem prover. Jhala et al. [17] also propose a similar method, where they reduce the constraint solving in the last phase to model checking of imperative programs. These approaches [16, 17, 31, 33] do support higher-order functions, but in a limited manner, in the sense that nested intersection types are not allowed. The difference between dependent types with/without intersections is like the one between context (or flow) sensitive/insensitive analyses. The former is more precise though it can be costly.[10] In general, nested intersection types are necessary to verify a program when function parameters are used more than once in different contexts. Indeed, as discussed in [25], several of the programs in Section 6 (e.g. `neg`, where the first argument of `twice` is used in two different contexts) require nested intersection types, and almost all the examples given by Kobayashi [20, 22] call for nested intersection types.

The limitation of our current prototype implementation is that the supported language features are limited. We believe that it is possible to extend our implementation to deal with data structures. In fact, the predicate abstraction introduced in Section 4 applies to data structures given an appropriate theorem prover. We expect the CEGAR part can also be extended, e.g. by restricting the properties on data structures to size properties, by treating data constructors as uninterpreted function symbols, etc.

Technically, most closest to ours is Terauchi's work [32]. In his method, candidates for dependent types are inferred from a finite unfolding of a source program, and then a fixedpoint computation algorithm is used to filter out invalid types. If the source program is not typable with the candidates for dependent types, the program is further unfolded and more candidates are collected. This cycle (which may diverge) is repeated until the source program is found to be well-typed or ill-typed. This is somewhat similar to the way our verification method works: abstraction types are inferred from an error trace (instead of an unfolding of a program), and then higher-order model checking (which also involves a fixedpoint computation) is applied to verify the abstract program. If the verification fails and an infeasible error path is found, the error path is used to infer more predicates, and this cycle is repeated. Thus, roughly speaking, our CEGAR phase corresponds to that of Terauchi to find candidates for dependent types, and our phases for predicate abstraction and higher-order model checking corresponds to Terauchi's fixedpoint computation phase. Advantages of

[10] Our method is an extreme case of context/flow sensitive analysis, which is sound and complete for programs with finite data domains.

program	S	O	C	abst	mc	cegar	total
intro1	27	2	1	0.00	0.00	0.00	0.01
intro2	29	2	1	0.00	0.00	0.00	0.00
intro3	30	2	1	0.00	0.00	0.00	0.00
sum	24	1	2	0.00	0.00	0.01	0.02
mult	31	1	2	0.01	0.00	0.02	0.03
max	42	2	1	0.00	0.00	0.03	0.03
mc91	32	1	2	0.01	0.04	0.02	0.07
ack	53	1	3	0.02	0.09	0.03	0.15
repeat	37	2	3	0.01	0.02	0.12	0.15
fhnhn	37	2	1	0.01	0.01	0.02	0.04
hrec	34	2	2	0.00	0.01	0.02	0.03
neg	47	2	1	0.01	0.01	0.01	0.03
apply	34	2	-	-	-	-	-
a-prod	70	2	4	0.07	0.06	0.08	0.22
a-cppr	149	2	6	0.32	2.82	0.26	3.40

program	S	O	C	abst	mc	cegar	total
a-init	96	2	5	0.16	0.18	0.38	0.73
a-max	70	2	5	2.34	2.01	0.43	4.78
l-zipunzip	81	2	3	0.03	0.08	0.02	0.12
l-zipmap	65	2	4	0.07	0.09	0.03	0.20
hors	64	2	2	0.00	0.00	0.00	0.01
e-simple	27	2	1	0.00	0.00	0.00	0.00
e-fact	55	2	2	0.00	0.01	0.00	0.01
r-lock	54	1	5	0.01	0.02	0.02	0.04
r-file	168	1	12	0.30	4.78	0.16	5.23
sum-e	26	1	0	0.00	0.00	0.00	0.00
mult-e	33	1	0	0.00	0.00	0.00	0.00
mc91-e	32	1	0	0.00	0.00	0.00	0.00
repeat-e	35	2	0	0.00	0.00	0.00	0.00
a-max-e	70	2	2	0.01	0.06	0.06	0.13
r-lock-e	54	1	0	0.00	0.00	0.00	0.00

Table 1. Results of preliminary experiments

ours are: (i) our method can generate an error path as a counterexample; there is no false alarm. On the other hand, a counterexample of Terauchi's method is an unfolding of a program, which may actually be safe. (ii) We infer predicates from an error trace, rather than from an unfolding of a program; From the latter, too many constraints are generated, especially for programs containing non-linear recursions. (iii) Our method can find dependent types constructed from arbitrary boolean combinations of the inferred predicates, while Terauchi's method only looks for dependent types constructed from the formulas directly generated by an interpolating theorem prover; thus, the success of the latter more heavily relies on the quality or heuristics of the underlying interpolating theorem prover. (iv) Because of the point (iii) above, our method (predicate abstraction + higher-order model checking) can also be used in a liquid type-like setting [31] where atomic predicates are given by a user. (v) Because of the point discussed in Remark 1, the combination of our predicate abstraction and higher-order model checking is strictly more powerful than Terauchi's approach (as long as suitable predicates are found). (vi) Our method can be extended to verify more general properties (expressed by the modal μ-calculus), by appealing to the results on higher-order model checking [24, 28].

7.3 Traditional Model Checking

Predicate abstraction and CEGAR have been extensively studied in the context of finite state or pushdown model checking [2–4, 8, 12–15]. Predicate abstraction has also been applied to the game-semantics-based model checking [1]. We are not, however, aware of previous work that applies predicate abstraction and CEGAR to higher-order model checking. As discussed in Section 1, the extension of predicate abstraction and CEGAR to higher-order model checking is non-trivial. One may think that defunctionalization [30] can be used to eliminate higher-order functions and apply conventional model checking. The defunctionalization however uses recursive data structures to represent closures, so that the resulting verification method is too imprecise, unless a clever abstraction technique for recursive data structures is available.

The three components of our verification method, predicate abstraction, higher-order model checking (TRECS), and CEGAR, may be seen as higher-order counterparts of the three components of SLAM [2–4]: C2BP, BEBOP, and NEWTON. Our use of dependency in abstraction types appears to subsume Ball et al.'s polymorphic predicate abstraction [3]. For example, the `id` function in [3] can be abstracted by using the abstraction type $x : \texttt{int}[\,] \rightarrow \texttt{int}[\lambda y.y = x]$.

There are a lot of studies to optimize predicate abstraction (especially for optimizing or avoiding the costly computation of abstract transition functions) in the context of conventional model checking [2, 27]. We have already borrowed some of the optimization techniques as mentioned in Section 6, and plan to adapt more techniques.

7.4 Abstract Interpretation

The combination of predicate abstraction and higher-order model checking may be viewed as a kind of abstract interpretation [9]. The abstract domain used for each functional value is defined by abstraction types, and predicate abstraction transforms a source program into an HBP whose semantics corresponds to the abstract semantics of the source program. Higher-order model checking then computes the abstract semantics. An advantage here is that thanks to the model checking algorithm [20] for higher-order recursion schemes, the computation of the abstract semantics is often much faster than a naive fixed-point computation (which is extremely costly for higher-order function values).

8. Conclusion

We have proposed predicate abstraction and CEGAR techniques for higher-order model checking, and implemented a prototype verifier. We believe that this is a new promising approach to automatic verification of higher-order functional programs. Optimization of the implementation and larger experiments are left for future work.

We conclude the paper with discussions on the scalability of our method. The current implementation is not scalable for large programs, but we expect that (after several years of efforts) we can eventually obtain a much more scalable verifier based on our method, for several reasons. First, the complexity of the model checking of higher-order recursion schemes is n-EXPTIME complete [28], but with certain parameters being fixed, the complexity is actually polynomial (linear, if restricted to safety properties) time in the size of a recursion scheme (or, the size of HBP). Furthermore, n-EXPTIME completeness is the *worst-case* complexity, and recent model checking algorithms [20, 23] do not immediately suffer from the n-EXPTIME bottleneck. Secondly, the implementation of the underlying higher-order model checker TRECS is premature, and there is a good chance for improvement. Thirdly, the current implementation of predicate abstraction and CEGAR is also quite naive. For example, the current implementation computes abstract programs eagerly. We expect that a good speed-up is obtained by computing abstract programs lazily.

Acknowledgment

We would like to thank members of our research group for discussion and comments. We would also like to thank anonymous referees for useful comments. This work is partially supported by Kakenhi 20240001.

References

[1] A. Bakewell and D. R. Ghica. Compositional predicate abstraction from game semantics. In *TACAS '09*, pages 62–76. Springer, 2009.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01*, pages 203–213. ACM, 2001.

[3] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems*, 27(2):314–343, 2005.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02*, pages 1–3. ACM, 2002.

[5] C. Barrett and C. Tinelli. CVC3. In *CAV '07*, volume 4590 of *LNCS*, pages 298–302. Springer, July 2007.

[6] D. Beyer, D. Zufferey, and R. Majumdar. CSIsat : Interpolation for LA+EUF (tool paper). In *CAV '08*, volume 5123 of *LNCS*, pages 304–308, July 2008.

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

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*, pages 84–96. ACM, 1978.

[10] W. Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20(2):95 – 207, 1982.

[11] C. Flanagan. Hybrid type checking. In *POPL '06*, pages 245–256. ACM, 2006.

[12] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV '97*, volume 1254 of *LNCS*, pages 72–83. Springer, June 1997.

[13] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL '10*, pages 471–482. ACM, 2010.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL '04*, pages 232–244. ACM, 2004.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, 2002.

[16] R. Jhala and R. Majumdar. Counterexample refinement for functional programs. Manuscript, available from `http://www.cs.ucla.edu/~rupak/Papers/CEGARFunctional.ps`, 2009.

[17] R. Jhala, R. Majumdar, and A. Rybalchenko. Refinement type inference via abstract interpretation. arXiv:1004.2884v1, 2010.

[18] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS '06*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[19] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS '02*, volume 2303 of *LNCS*, pages 205–222. Springer, 2002.

[20] N. Kobayashi. Model-checking higher-order functions. In *PPDP '09*, pages 25–36. ACM, 2009.

[21] N. Kobayashi. TRecS. `http://www.kb.ecei.tohoku.ac.jp/~koba/trecs/`, 2009.

[22] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *POPL '09*, pages 416–428. ACM, 2009.

[23] N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *FoSSaCS '11*. Springer, 2011.

[24] N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS '09*, pages 179–188. IEEE, 2009.

[25] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. An extended version, available from `http://www.kb.ecei.tohoku.ac.jp/~uhiro/`, 2011.

[26] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL '10*, pages 495–508. ACM, 2010.

[27] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

[28] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS '06*, pages 81–90. IEEE, 2006.

[29] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL '11*, pages 587–598. ACM, 2011.

[30] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference - Volume 2*, pages 717–740. ACM, 1972.

[31] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*. ACM, 2008.

[32] T. Terauchi. Dependent types from counterexamples. In *POPL '10*, pages 119–130. ACM, 2010.

[33] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288. ACM, 2009.

[34] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *APLAS '10*, volume 6461 of *LNCS*, pages 312–327. Springer, October/December 2010.

[35] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM, 1999.