Towards a Scalable Software Model Checker for Higher-Order Programs

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

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

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

Abstract

In our recent paper, we have shown how to construct a fully-automated program verification tool (so called a "software model checker") for a tiny subset of functional language ML, by combining higher-order model checking, predicate abstraction, and CEGAR. This can be viewed as a higher-order counterpart of previous software model checkers for imperative languages like BLAST and SLAM. The naïve application of the proposed approach, however, suffered from scalability problems, both in terms of efficiency and supported language features. To obtain more scalable software model checkers for full-scale functional languages, we propose a series of optimizations and extensions of the previous approach. Among others, we introduce (i) selective CPS transformation, (ii) selective predicate abstraction, and (iii) refined predicate discovery as optimization techniques; and propose (iv) functional encoding of recursive data structures and control operations to support a larger subset of ML. We have implemented the proposed methods, and obtained promising results.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Higher-Order Model Checking, Predicate Abstraction, Abstraction Refinement, Dependent Types, CPS Transformation

1. Introduction

In our recent paper [15], we have shown how to construct a fully-automated program verification tool (so called a "software model checker") for a tiny subset of functional language ML, a simply-typed lambda calculus with recursion and integers. The framework is an extension of higher-order model checker (more precisely, the model checker for higher-order recursion scheme) for infinite data domains such as integers, obtained by combining the techniques of predicate abstraction [8] and counterexample-guided abstraction refinements (CEGAR) [1, 6]. This can be viewed as a higher-order counterpart of previous software model checkers for imperative languages like BLAST [9] and SLAM [1]. We have implemented a verification tool, MoCHi, based on the framework.

The naïve application of the proposed approach, however, suffered from scalability problems, both in terms of efficiency and supported language features. For example, the previous framework does not support recursive data structures and our implementation based on the framework does not support control operations. To explain the problem of the previous framework/implementation, let us first review our previous framework.

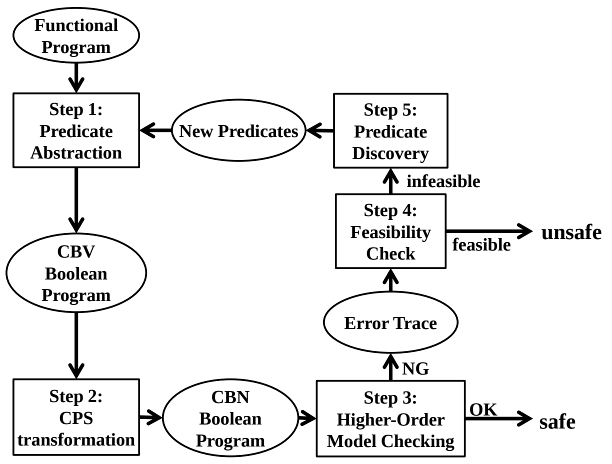
Functional Program
Step 1: Predicate Abstraction
New Predicates
Step 5: Predicate Discovery
infeasible
Step 4: Feasibility Check
feasible
unsafe
CBV Boolean Program
Error Trace
NG
Step 2: CPS transformation
CBN Boolean Program
Step 3: Higher-Order Model Checking
OK
safe

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

Our previous verification framework [15] is based on predicate abstraction and CEGAR for a higher-order model checker. Figure 1 shows the overall structure of our previous method. In Step 1, an input program, written in a simply typed call-by-value lambda calculus with recursion and integers, is abstracted to a higher-order boolean program by predicate abstraction. The abstracted program is verified by a higher-order model checker, where models are described by higher-order recursion schemes, in Step 3. Higher-order recursion schemes [18] can be viewed as a simply typed call-by-name lambda calculus with finite data domains and recursion. To resolve the gap between the evaluation strategies of higher-order recursion schemes and higher-order boolean programs, we translate the call-by-value (CBV) program into a call-by-name (CBN) program in Step 2. If the abstracted program is safe (i.e., assertions never fail), the original functional program is also safe. If not, by using a counterexample produced by the higher-order model checker, we check whether the original program is in fact unsafe or the abstraction is too coarse in Step 4. If the latter, we discover new predicates in Step 5. We repeat these steps until we find whether the program is safe or not.

In this paper, we address several limitations of our previous work. We discuss each problem and explain our approach below.

1. Continuation-passing style (CPS) transformation in Step 2 caused an efficiency problem for the previous version of MoCHi.

```
letrec check x f = f x; check (x+1) f
let f x = assert (x >= 0)
let main n = check n f
```

Figure 2. Example Program of Selective CPS Transformation

```
let add x y = x + y
letrec sum x = if x<=0 then 0 else add x (sum(x-1))
let main x = assert (sum x >= x)
```

Figure 3. Example Program of Selective Predicate Abstraction

As stated above, our approach requires CPS transformation to resolve the gap between the evaluation strategies of higher-order recursion schemes and higher-order boolean programs. However, CPS transformation significantly increases the order[1] of programs. Since the complexity of model checking of higher-order recursion scheme is n-EXPTIME complete for order-n recursion scheme, the increase of the order of an input is crucial to the verification time. For example, consider the program shown in Figure 2. The above program is translated into the following program by a naïve call-by-value CPS transformation [20].

```
letrec check x k1 =
  k1 (λf.λk2.f x (λ_.check (x+1) (λg.g f k2)))
let f x k = assert (x >= 0); k ()
let main n k = check n (λg. g f k)
```

To preserve the evaluation order of an original call-by-value program, each function of the translated call-by-name program takes a continuation and passes its return value to the continuation. Note here that the order of the translated program has increased to 5. The continuation `k1` of `check` can actually be omitted however, since a partial application of `check` in the original program never causes side effects including an assertion failure. We can, therefore, obtain the following order-3 program.

```
letrec check x f k = f x (λ_.check (x+1) f k)
let f x k = assert (x >= 0); k ()
let main n k = check n f k
```

Our CPS transformation avoids such unnecessary insertion of continuations. We formalize this transformation, called *selective CPS transformation*.

2. The main bottleneck of the previous version of MoCHi was the predicate abstraction and discovery (Steps 1 and 5), rather than higher-order model checking. Our previous method for predicate abstraction tried to abstract *every* function in the program. As a result, verification of a program did not succeed until an appropriate set of predicates is found for every function. For example, consider the program shown in Figure 3. The program can be successfully verified if we abstract the program by using predicates $\lambda y.\, y \geq 0$ for the second argument of `add` and $\lambda r.\, r \geq x$ for the return values of `add` and `sum`. The following is the abstracted program obtained by using the above predicates.

```
let add () b = b
letrec sum () = if * then true
  else add () (if sum () then true else *)
let main x = assert (sum ())
```

Here, * means a non-deterministic boolean value. The argument `b` denotes whether `y` ≥ 0 holds or not. In the previous paper [15], we have adapted CEGAR to find such predicates, but the technique is necessarily heuristic, and often fails. If we cannot find the predicates for `add`, we get the following abstracted program.

```
let add () () = ()
letrec sum u = if * then true
  else let b = sum () and u = add () () in *
let main x = assert (sum ())
```

Since the return value of `add` is non-informative, the abstraction is too coarse for proving the safety of the original program.
To reduce the burden to the predicate discovery phase, we introduce a refinement of predicate abstraction called *selective predicate abstraction*. As the name suggests, the selective predicate abstraction applies predicate abstraction to only a certain set of functions, and avoids abstraction of the other functions by inlining them. The selective predicate abstraction generates the following safe program by using only the predicate $\lambda r.\, r \geq x$ for the return values of `sum` and inlining `add`.

```
letrec sum () =
  if * then true else if sum() then true else *
let main () = assert (sum ())
```

In this way the selective predicate abstraction improves the precision of abstraction and reduces the number of CEGAR iterations.

```
letrec copy x = if x=0 then 0 else 1 + copy (x-1)
let main n = assert (copy (copy n) = n)
```

Figure 4. Example Program for Refined Predicate Discovery

3. Even with the selective predicate abstraction, the previous version of MoCHi sometimes failed to discover an appropriate set of predicates. For example, consider the program shown in Figure 4. The program can be successfully verified if we abstract the program by using the predicate $r = x$ on the argument x and the return value r of `copy`. The old version of MoCHi, however, continued to find too specific predicates such as $x = 0 \wedge r = 0, x = 1 \wedge r = 1$, ..., infinitely. This is due to the too much path- and context-sensitivity of MoCHi. In Step 5, MoCHi first prepares a template of a refinement type for *each* function call in the spurious error path. For example, if there are two calls of the `copy` function, we prepare two templates $x : \{v : \mathbf{int} \mid P_1(v)\} \Rightarrow \{r : \mathbf{int} \mid Q_1(x, r)\}$ and $x : \{v : \mathbf{int} \mid P_2(v)\} \Rightarrow \{r : \mathbf{int} \mid Q_2(x, r)\}$. MoCHi then generates and solves constraints on the predicate variables in the template. The advantages of the approach are the context-sensitivity (different types can be assigned to different calls) and the existence of a complete algorithm (modulo certain assumptions) to solve the constraints, but the disadvantage is that the inferred constraints are often too specific. To address this problem, we refine our predicate discovery method by merging information about multiple calls of the same function as much as possible; in the example above, we merge P_1 with P_2, and Q_1 with Q_2 as long as the resulting constraints are satisfiable. Combined with some heuristic for using an underlying theorem prover, the refined method tends to discover a better set of predicates, as confirmed by experiments.

4. Another limitation was that many important language features such as recursive data structures and exceptions were not supported. Predicate abstraction can in principle be applied to recursive data structures, but there is not a good practical (interpolating) theorem prover that can be used for finding appropriate predicates on recursive data structures.[2] To support recursive data structures without relying on an interpolating theorem prover for them, we encode recursive data structures using higher-order functions and reduce verification of programs manipulating data structures to that of programs manipulating integers. For example, a list is encoded to a function that maps an index i to the i-th element. Similarly, we support control operations (e.g., exceptions and `call/cc`) by encoding them using higher-order functions.

[1] The order of program t is the maximum order of the types of functions in t. The order of type τ is defined by $order(\mathbf{int}) = 0$, $order(\tau_1 \to \tau_2) = max(order(\tau_1) + 1, order(\tau_2))$.

[2] One approach would be to treat data constructors as uninterpreted function (UF) symbols, and use an interpolating theorem prover that handles UF. It cannot, however, be used for finding inductive predicates such as “a list ℓ is a sequence of the form $(ab)^*$”.

The rest of this paper is organized as follows. We first formalize the source language of verification in Section 2. Section 3 formalizes the selective predicate abstraction and the selective CPS transformation. Section 4 shows the refined predicate discovery. Section 5 describes the language extensions for recursive data structures and control operations. Section 6 reports experiments. Section 7 discusses related work and Section 8 concludes the paper. For the space restriction, proofs are omitted, which will be available in the first author's forthcoming PhD thesis.

2. Source Language

In this section, we introduce the source language of our verification method. The source language is a simply-typed call-by-value higher-order functional language with recursion and integers (a la "PCF") with the syntax defined by:

$$\begin{array}{lcl}
D \text{ (programs)} & ::= & \{f_1 = v_1, \dots, f_n = v_n\} \\
t \text{ (terms)} & ::= & n \mid x \mid \lambda x.\, t \mid t_1\ t_2 \mid \mathrm{op}(t_1, t_2) \\
 & \mid & \textbf{if0}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid \textbf{fail} \\
v \text{ (values)} & ::= & n \mid x \mid \lambda x.\, t \\
\tau \text{ (types)} & ::= & \textbf{int} \mid \tau_1 \to \tau_2
\end{array}$$

The meta-variable op ranges over the set of operators over integers. The expressions are standard except that there is a primitive **fail** that aborts a program. We use true and false as aliases of 0 and 1, and we write $\mathbf{assert}(t)$ for $\textbf{if0}\ t\ \textbf{then}\ 0\ \textbf{else fail}$, $\textbf{let}\ x = t_1\ \textbf{in}\ t_2$ for $(\lambda x.\, t_2)\ t_1$, $(t_1\ \mathrm{op}\ t_2)$ for $\mathrm{op}(t_1, t_2)$. We assume that (i) a program is well-typed in the standard simple type system, (ii) every function in a program D has a function type, and (iii) D contains a distinguished function symbol $\mathit{main} \in \{f_1, \dots, f_n\}$ whose type is $\textbf{int} \to \textbf{int}$. The goal of our verification is to check whether $\mathit{main}\ n \not\longrightarrow^*_D \textbf{fail}$ for all integer n.

3. Optimizations

This section introduces optimizations for each phase of CPS transformation (Step 2) and predicate abstraction (Step 1) in Figure 1.

3.1 Selective CPS transformation

This section formalizes selective CPS transformation for abstracted programs. As stated in Section 1, the idea of the selective CPS transformation is to distinguish whether a continuation parameter should be inserted to each expression or not based on whether it has a side-effect. When a function application has no side-effects, we need not insert a continuation. Here, **fail**, non-deterministic branch, and non-termination are considered as side-effects. A similar transformation has been proposed by Nielsen [17]. The transformation does not fit our purpose to translate call-by-value programs into equivalent call-by-name programs. A non-terminating program may be transformed into a terminating (call-by-name) program by his transformation.

We first define the source language (and the target language) of selective CPS transformation. The language is the same as the one in Section 2 except: (i) the set of terms is extended with non-deterministic branch $t_1 \blacksquare t_2$, and (ii) a label $\ell \in \{C, N\}$ is attached to each function type, like $\tau_1 \to^\ell \tau_2$. The labels indicate whether we should insert a continuation or not.

$$\Gamma \vdash n : \textbf{int}, N \rightsquigarrow n \qquad \text{(CPS-CONST)}$$

$$\Gamma, x : \tau \vdash x : \tau, N \rightsquigarrow x \qquad \text{(CPS-VAR)}$$

$$\frac{\Gamma \vdash t_1 : \textbf{int}, N \rightsquigarrow t_1' \qquad \Gamma \vdash t_2 : \textbf{int}, N \rightsquigarrow t_2'}{\Gamma \vdash \mathrm{op}(t_1, t_2) : \textbf{int}, N \rightsquigarrow \mathrm{op}(t_1', t_2')} \qquad \text{(CPS-OPN)}$$

$$\frac{\Gamma \vdash t_1 : \textbf{int}, \ell_1 \rightsquigarrow t_1' \qquad \Gamma \vdash t_2 : \textbf{int}, \ell_2 \rightsquigarrow t_2'}{\Gamma \vdash \mathrm{op}(t_1, t_2) : \textbf{int}, N \rightsquigarrow @^{\ell_1}(t_1', \lambda x_1.\, @^{\ell_2}(t_2', \lambda x_2.\, \mathrm{op}(x_1, x_2)))} \qquad \text{(CPS-OPC)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2, N \rightsquigarrow t'}{\Gamma \vdash \lambda x.\, t : \tau_1 \to^N \tau_2, N \rightsquigarrow \lambda x.\, t'} \qquad \text{(CPS-ABSN)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2, \ell \rightsquigarrow t'}{\Gamma \vdash \lambda x.\, t : \tau_1 \to^C \tau_2, N \rightsquigarrow \lambda x.\, \lambda k.\, @^\ell(t', k)} \qquad \text{(CPS-ABSC)}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to^N \tau, N \rightsquigarrow t_0' \quad \Gamma \vdash t_1 : \tau_1, N \rightsquigarrow t_1'}{\Gamma \vdash t_0\ t_1 : \tau, N \rightsquigarrow t_0'\ t_1'} \qquad \text{(CPS-APPN)}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to^\ell \tau, \ell_0 \rightsquigarrow t_0' \qquad \Gamma \vdash t_1 : \tau_1, \ell_1 \rightsquigarrow t_1'}{\Gamma \vdash t_0\ t_1 : \tau, C \rightsquigarrow \lambda k.\, @^{\ell_0}(t_0', \lambda x_0.\, @^{\ell_1}(t_1', \lambda x_1.\, @^\ell(x_0\ x_1, k)))} \qquad \text{(CPS-APPC)}$$

$$\Gamma \vdash \textbf{fail} : \tau, C \rightsquigarrow \lambda k.\, \textbf{fail} \qquad \text{(CPS-FAIL)}$$

$$\frac{\Gamma \vdash t_1 : \textbf{int}, N \rightsquigarrow t_1' \quad \Gamma \vdash t_2 : \tau, N \rightsquigarrow t_2' \quad \Gamma \vdash t_3 : \tau, N \rightsquigarrow t_3'}{\Gamma \vdash \textbf{if0}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 : \tau, N \rightsquigarrow \textbf{if0}\ t_1'\ \textbf{then}\ t_2'\ \textbf{else}\ t_3'} \qquad \text{(CPS-IFN)}$$

$$\frac{\Gamma \vdash t_1 : \textbf{int}, \ell_1 \rightsquigarrow t_1' \quad \Gamma \vdash t_2 : \tau, \ell_2 \rightsquigarrow t_2' \quad \Gamma \vdash t_3 : \tau, \ell_3 \rightsquigarrow t_3'}{\begin{array}{c}\Gamma \vdash \textbf{if0}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 : \tau, C \rightsquigarrow \lambda k.\, \mathit{App}^{\ell_1}(t_1', \\ \lambda m.\, \textbf{if0}\ m\ \textbf{then}\ @^{\ell_2}(t_2', k)\ \textbf{else}\ @^{\ell_3}(t_3', k))\end{array}} \qquad \text{(CPS-IFC)}$$

$$\frac{\Gamma \vdash t_1 : \tau, \ell_1 \rightsquigarrow t_1' \qquad \Gamma \vdash t_2 : \tau, \ell_2 \rightsquigarrow t_2'}{\Gamma \vdash t_1 \blacksquare t_2 : \tau, C \rightsquigarrow @^{\ell_1}(t_1', k) \blacksquare @^{\ell_2}(t_2', k)} \qquad \text{(CPS-BR)}$$

$$\frac{\begin{array}{c} v_i \text{ is of the form } \lambda x_1.\, \lambda x_2.\, \dots \lambda x_j.\, t_i \qquad t_i \text{ is not of the form } \lambda x.\, t_i' \\ \Gamma(f_i) \text{ is of the form } \tau_{i1} \to^{\ell_1} \tau_{i2} \to^{\ell_2} \dots \to^{\ell_{i(j-1)}} \tau_{ij} \to^C \tau \\ \Gamma \vdash v_i : \Gamma(f_i), N \rightsquigarrow v_i' \quad \text{for each } i \in \{1, \dots, n\} \end{array}}{\Gamma \vdash \{f_1 = v_1, \dots, f_n = v_n\} \rightsquigarrow \{f_1 = v_1', \dots, f_n = v_n'\}} \qquad \text{(CPS-PROG)}$$

Figure 5. Selective CPS transformation

Below, we formalize the selective CPS as a type-based transformation. Figure 5 shows the rules of the selective CPS transformation. In the figure, $@^\ell(-,-)$ is an operation defined as $@^N(t, k) = k\ t$ and $@^C(t, k) = t\ k$. The relation $\Gamma \vdash t : \tau, \ell \rightsquigarrow t'$ means that a term t is translated to a term t' by using a type τ, under the assumption that (i) each free variable x of t has been transformed using the type $\Gamma(x)$, (ii) t may have a side-effect if $\ell = C$, and (iii) t has no side-effect if $\ell = N$. The relation $\Gamma \vdash D \rightsquigarrow D'$ means that a program D is translated to a program D' according to Γ. The rules are designed in such a way that transformed term t' has type $\llbracket\tau\rrbracket$ if $\Gamma \vdash t : \tau, N \rightsquigarrow t'$ holds, and the transformed term t' has type $(\llbracket\tau\rrbracket \to X) \to X$ if $\Gamma \vdash t : \tau, C \rightsquigarrow t'$ holds, where X is the answer type and $\llbracket\tau\rrbracket$ is defined by:

$$\llbracket\textbf{int}\rrbracket = \textbf{int} \qquad \llbracket\tau_1 \to^N \tau_2\rrbracket = \llbracket\tau_1\rrbracket \to \llbracket\tau_2\rrbracket$$
$$\llbracket\tau_1 \to^C \tau_2\rrbracket = \llbracket\tau_1\rrbracket \to (\llbracket\tau_2\rrbracket \to X) \to X$$

Thus, a continuation should be inserted only for functions of type $\tau_1 \to^C \tau_2$, not for functions of type $\tau_1 \to^N \tau_2$.

In the rule CPS-ABSC, a term $\lambda x.\, t$ is transformed in a standard way, i.e. a continuation parameter k is inserted and t', the CPS version of t, is applied to k. On the other hand, in the rule CPS-ABSN, no continuation parameter is inserted and direct style is preserved. The rules for applications are similar. In the rule CPS-APPC, a continuation is inserted, but not in the rule CPS-APPN. In the rule CPS-BR, a term $t_1 \blacksquare t_2$ should be transformed with C and a continuation is needed because we need to treat non-deterministic branch as a side-effect. In the rule CPS-PROG, since an application of a top-level function may cause a side-effect, non-termination, the function type for the last argument is annotated with C. For the sake of simplicity, regardless of whether functions are recursive, we consider all the top-level functions may not terminate conservatively. We can avoid redundant insertion of continuations by some termination analysis.

As an example, Figure 6 shows the selective CPS transformation of $\lambda x.\, \lambda y.\, \mathtt{assert}(x = y)$ with type $\textbf{int} \to^N \textbf{int} \to^C \textbf{int}$ for (a) and $\textbf{int} \to^C \textbf{int} \to^C \textbf{int}$ for (b). The transformation of

(a)
$$\dfrac{\dfrac{\dfrac{\Gamma_{xy} \vdash \texttt{assert} : \mathbf{bool} \to^{C} \mathbf{unit}, N \leadsto \texttt{assert}_{\texttt{CPS}} \qquad \vdots}{\Gamma_{xy} \vdash t : \mathbf{unit}, C \leadsto \lambda k.\, t'\; k}}{\Gamma_x \vdash \lambda y.\, t : \mathbf{int} \to^{C} \mathbf{unit}, N \leadsto \lambda y.\, \lambda k.\, t'\; k}}{\Gamma \vdash \lambda x.\, \lambda y.\, t : \mathbf{int} \to^{N} \mathbf{int} \to^{C} \mathbf{unit}, N \leadsto \lambda x.\, \lambda y.\, \lambda k.\, t'\; k}$$

(b)
$$\dfrac{\dfrac{\vdots}{\Gamma_x \vdash \lambda y.\, t : \mathbf{int} \to^{C} \mathbf{unit}, N \leadsto \lambda y.\, \lambda k.\, t'\; k}}{\begin{array}{l}\Gamma \vdash \lambda x.\, \lambda y.\, t : \mathbf{int} \to^{C} \mathbf{int} \to^{C} \mathbf{unit}, N \leadsto \\ \qquad\qquad \lambda x.\, \lambda k'.\, k'\; (\lambda y.\, \lambda k.\, t'\; k)\end{array}}$$

Figure 6. Transformations of $\lambda x.\, \lambda y.\, \texttt{assert}(x = y)$. ($t = \texttt{assert}(x = y)$, $t' = \texttt{assert}_{\texttt{CPS}}(x = y)$, $\Gamma_x = \Gamma, x : \mathbf{int}$. $\Gamma_{xy} = \Gamma_x, y : \mathbf{int}$. $\texttt{assert}_{\texttt{CPS}}\; b\; k$ asserts b and returns $k\; ()$)

$\lambda y.\, \texttt{assert}(x = y)$ in (b) is the same as the one in (a). In (a), there is no need to insert a continuation parameter after x. On the other hand, in (b), a continuation parameter is inserted.

When we infer types and labels, we attach label N to function types as much as possible. Consider function `check` shown in Figure 2. By the rule CPS-PROG, the type of `check` τ must be of the form $\mathbf{int} \to^{\ell} (\mathbf{int} \to^{\ell'} \mathbf{unit}) \to^{C} \mathbf{unit}$. Since there is no constraint such that $\tau = \tau_1 \to^{C} \tau_2$, we instantiate ℓ to label N.

We state properties of the selective CPS transformation. The first theorem below states that we can transform arbitrary simply-typed programs by the selective CPS transformation. The second theorem states that the selective CPS transformation is correct in the sense that the transformed program is reduced to the same value as the original program.

Theorem 3.1. *Suppose D is typable in Γ. There exists Γ' and D' such that $\Gamma = \texttt{Elim}(\Gamma')$ and $\Gamma' \vdash D \leadsto D'$, where $\texttt{Elim}(\Gamma)$ is the type environment obtained from Γ by removing annotations.*

Theorem 3.2 (Correctness of CPS Transformation)**.** *If $\Gamma \vdash D \leadsto D'$, then the following holds: $main\ n \longrightarrow_D^* \mathbf{fail}$ if and only if $main\ n\ (\lambda x.\, x) \longrightarrow_{D'}^* \mathbf{fail}$.*

3.2 Selective Predicate Abstraction

We introduce an optimization technique for predicate abstraction, called selective predicate abstraction.

The idea of selective predicate abstraction is to apply predicate abstractions only to a certain set of functions and inline the other functions. For example, the program in Figure 3 can be verified by abstracting only `sum` and `main` as stated in Section 1.

We formalize selective predicate abstraction as an extension of the previous predicate abstraction. In the previous framework, predicate abstraction is defined as the relation $\Gamma \vdash_{\texttt{NS}} t : \sigma \leadsto t'$ which means that t is abstracted to t' by using the abstraction type σ under the assumption that each free variable x of t has been abstracted using the abstraction type $\Gamma(x)$. Abstraction types are types to express which predicate should be used to abstract each value. For example, an abstraction type $\mathbf{int}[P_1, \ldots, P_n]$ means that a value v of that type should be abstracted to a tuple $(b_1, \ldots, b_n)$, where b_i denotes whether $P_i(v)$ holds or not. See our previous paper [15] for details. We extend the predicate abstraction relation by adding the set of inlined functions E like $\Gamma \mid E \vdash t : \sigma \leadsto t'$. Here, functions in E are not abstracted and are inlined in the process of abstraction. An alternative approach would be to inline all the functions in E first, and then apply ordinary predicate abstraction [15]; we prefer the formalization below (that inline functions on-demand) as it seems more conveninent for further optimizations such as memoization of abstractions.

$$\dfrac{\begin{array}{c} x \notin dom(E) \qquad \Gamma(x) = (y_1 : \sigma_1 \to \cdots \to y_n : \sigma_n \to \sigma) \\ \begin{array}{c}\Gamma, y_1 : \sigma_1, \ldots, y_{i-1} : \sigma_{i-1} \mid E \vdash \\ v_i : [v_1/y_1, \ldots, v_{i-1}/y_{i-1}]\sigma_i \leadsto e_i\end{array} \quad \text{for each } i \in \{1, \ldots, n\} \end{array}}{\Gamma \mid E \vdash x\, \widetilde{v} : [\widetilde{v}/\widetilde{y}]\sigma \leadsto \mathbf{let}\ y_1 = e_1\ \mathbf{in} \cdots \mathbf{let}\ y_n = e_n\ \mathbf{in}\ x\, \widetilde{y}}$$
(A-APP)

$$\dfrac{E(x) = \lambda \widetilde{x}.\, e \qquad \Gamma \mid E \vdash [\widetilde{v}/\widetilde{x}]e : \sigma \leadsto e'}{\Gamma \mid E \vdash x\, \widetilde{v} : \sigma \leadsto e'} \text{ (A-APPEXP)}$$

$$\dfrac{\begin{array}{c} \Gamma \mid E \vdash v_i : \Gamma(f_i) \leadsto v_i' \qquad \text{for each } i \text{ s.t. } f_i \notin dom(E) \\ E \subseteq \{f_1 = v_1, \ldots, f_n = v_n\} \qquad dom(\Gamma) \cap dom(E) = \emptyset \end{array}}{\Gamma \mid E \vdash \{f_1 = v_1, \ldots, f_n = v_n\} \leadsto \{f_i = v_i' \mid f_i \notin dom(E)\}}$$
(A-PROG)

Figure 7. Selective Predicate Abstraction

$$\dfrac{\vdots \qquad \dfrac{\dfrac{\vdots}{\Gamma' \mid E \vdash x + s : \mathbf{int}[\lambda r.\, r \geq x] \leadsto \mathbf{if0}\ s\ \mathbf{then}\ \mathit{true}\ \mathbf{else} *}}{\Gamma' \mid E \vdash \texttt{add}\; x\; s : \mathbf{int}[\lambda r.\, r \geq x] \leadsto \mathbf{if0}\ s\ \mathbf{then}\ \mathit{true}\ \mathbf{else} *}}{\begin{array}{l}\Gamma \mid E \vdash \mathbf{let}\ s = \texttt{sum}\ (x-1)\ \mathbf{in}\ \texttt{add}\; x\; s : \mathbf{int}[\lambda r.\, r \geq x] \leadsto \\ \qquad\qquad \mathbf{let}\ s = \texttt{sum}\ ()\ \mathbf{in}\ \mathbf{if0}\ s\ \mathbf{then}\ \mathit{true}\ \mathbf{else} *\end{array}}$$

Figure 8. Abstraction of $\texttt{add}\, x\ (\texttt{sum}\, (x-1))$. ($\Gamma = x : \mathbf{int}[], \texttt{sum} : (y : \mathbf{int}[] \to \mathbf{int}[\lambda r.\, r \geq y])$, $x > 0$. $\Gamma' = \Gamma$, $s : \mathbf{int}[\lambda s.\, s \geq x - 1]$. $E = \{\texttt{add} = \lambda x.\, \lambda y.\, x + y\}$, * is a syntactic sugar for *true* ■ *false*.)

Figure 7 shows the key rules of the selective predicate abstraction. Other rules are the same as the previous predicate abstraction relation [15] except that the set of inlined functions E is added. The relation $\Gamma \mid E \vdash t : \tau \leadsto t'$ means that t is abstracted to t' by using the abstraction type τ under the assumption that each free variable $x \in dom(\Gamma)$ of t has been abstracted using the abstraction type $\Gamma(x)$ and each free variable $x \in dom(E)$ of t is inlined. If E is empty, the selective predicate abstraction relation is the same as the previous predicate abstraction.

Figure 8 shows a part of the abstraction of the program shown in Figure 3 with $E = \{\texttt{add} = \lambda x.\, \lambda y.\, x + y\}$. Due to the restriction of the source language of previous predicate abstraction [15], we translated the term $\texttt{add}\, x\ (\texttt{sum}\ (x-1))$ to $\mathbf{let}\ s = \texttt{sum}\ (x - 1)\ \mathbf{in}\ \texttt{add}\ x\ s$. In the abstraction of $\texttt{add}\ x\ s$, `add` is inlined since the definition of `add` is in E. On the top of the derivation, $x + s$ is abstracted to $\mathbf{if0}\ s\ \mathbf{then}\ \mathit{true}\ \mathbf{else}\ (\mathit{true}$ ■ $\mathit{false})$ since $s \geq x$ holds if translated s is true (that denotes $s \geq x - 1$), and we do not know whether $s \geq x$ otherwise.

For some D and E, there is D' such that $\Gamma \mid E \vdash D \leadsto D'$ as long as there is no cyclic definitions in E like $\{f = \lambda x.\, g\ x,\ g = \lambda y.\, f\}$. Therefore, one way to decide E is to find a maximal set of the functions that has no cyclic definitions. Even if E has some functions which are recursive in D, there is D' such that $\Gamma \mid E \vdash D \leadsto D'$ in some cases. For example, the following program can be abstracted with $E = \{\texttt{odd} = v_{\texttt{odd}}\}$.

```
letrec even x = if x = 0 then true else odd (x-1)
   and odd x = if x = 0 then false else even (x-1)
let main () = assert (even (n+n))
```

The following is the abstraction of the program with $\Gamma = \texttt{even} : \mathbf{int}[P_e] \to \mathbf{bool}, \texttt{main}{:}\mathbf{int}[] \to \mathbf{unit}$ where $P_e = \lambda x.\, (x \bmod 2 = 0)$.

```
letrec even b =
  if (if b then * else false) then true
  else if (if b then false else *) then false
  else even b
let main () = assert (even b)
```

The abstracted program is safe. For comparison, consider the case where $\Gamma = \texttt{even} : \mathbf{int}[P_e] \to \mathbf{bool}, \texttt{odd} : \mathbf{int}[] \to \mathbf{bool}, \texttt{main} : \mathbf{int}[] \to \mathbf{unit}$ and $E = \emptyset$ (which corresponds to our previous approach [15] where the predicates for `odd` have not been found).

```
letrec copy x = if x=0 then 0 else 1 + copy (x-1)
let main n = assert (copy n = n)
```

Figure 9. A Simplified Version of the Program in Figure 4

```
 letrec even b = if (if b then * else false)
                 then true else odd ()
    and odd () = if * then false else even *
 let main () = assert (even b)
```

Since the abstraction of odd is too coarse, the abstracted even returns a non-deterministic booleans. Thus, the abstracted program is unsafe.

We state properties of the selective predicate abstraction. The theorem below states that the predicate abstraction is sound in the sense that if the original program fails, so does its abstraction.

Theorem 3.3 (Soundness)**.** *If* $\Gamma \mid E \vdash D_1 \rightsquigarrow D_2$*, and* $main\ n \longrightarrow^*_{D_1}$ **fail**, *then* $main\ n \longrightarrow^*_{D_2}$ **fail**.

The proposition below states that selective predicate abstraction is more precise compared to our previous predicate abstraction. Here, $\Gamma \vdash_{\mathtt{NS}} D : \tau \rightsquigarrow D_1$ is the predicate abstraction relation in our previous paper [15].

Proposition 3.4. *If* $\Gamma \vdash_{\mathtt{NS}} D : \tau \rightsquigarrow D_1, \Gamma \mid E \vdash D \rightsquigarrow D_2$*, and* $main\ n \longrightarrow^*_{D_2}$ **fail**, *then* $main\ n \longrightarrow^*_{D_1}$ **fail**.

4. Predicate Discovery

In this section, we propose an extension of our previous predicate discovery method for higher-order programs used in MoCHi [15]. First, we briefly overview the previous method in Section 4.1 and then discuss its limitation in Section 4.2. Section 4.3 explains the extension of the method, which remedies the limitation.

4.1 Previous Method

In MoCHi, predicates for abstracting each term of a given program are specified as a kind of dependent types called abstraction types. MoCHi infers abstraction types automatically in a counterexample-guided manner (recall Figure 1): In a CEGAR iteration of MoCHi, if the predicate abstraction at that point is not precise enough to show the safety of the original program, an error path of the abstracted program is returned as a result of higher-order model checking. If the abstract error path is infeasible (i.e., not a genuine path of the original program), MoCHi generates a straightline higher-order program (SHP) that is safe if and only if the abstract error path is infeasible. MoCHi then uses an existing method [24] to infer refinement types that witness the safety of the SHP. Here, to make the inference context-sensitive and complete as discussed in Section 1, MoCHi ensures the generated SHP to be linear (i.e., each function is called exactly once) and recursion-free by duplicating and renaming the functions called multiple times in the infeasible error path (see [15] for more details). Finally, MoCHi extracts abstraction types from the refinement types, which contain precise enough predicates to refute the infeasible error path.

The key ingredient of the above predicate discovery procedure is the refinement type inference method [24], which consists of two steps: constraint generation and solving. We review the two steps respectively in Sections 4.1.1 and 4.1.2.

4.1.1 Constraint Generation

Given a spurious error path, we can construct a SHP D which is typable under a refinement type system (see, for example, [24] for the definition of the system). For example, let us consider the program in Figure 9, which is a simplified version of the one in Figure 4. In the course of its verification, we may obtain the following SHP $D_{\mathtt{copy}}$:

```
 let copy1 x = assume (x<>0); 1 + copy2 (x-1)
 let copy2 x = assume (x=0); 0
 let main n = assume (copy1 n <> n); fail
```

Here, assume e evaluates e and proceeds to the next command only if e evaluates to true. The SHP corresponds to an infeasible error path where the else- and the then-branches of copy are respectively taken in the first and the second function call of copy, and the assertion in the main function fails. Note here that $D_{\mathtt{copy}}$ is safe (i.e., fail is not reachable), and hence is typable under the refinement type system.

From a SHP D, we generate Horn-clause-like constraints which are satisfiable if and only if D is typable. To this end, for each function in D, we prepare a refinement type template with predicate variables, which act as placeholders of refinement predicates to be inferred. We then generate a typing derivation for D under the type environment that associates each function with its type template. Horn-clause-like constraints on the predicate variables are then extracted from the derivation. Since the SHP D is linear and recursion-free, generated constraints are non-recursive. This is desirable since constraint solving of non-recursive Horn clauses over decidable underlying theories (e.g., linear arithmetic) is decidable. For the running example $D_{\mathtt{copy}}$, we use the following templates:[3] $\mathtt{copy1} : (x : \mathbf{int} \to \{\nu : \mathbf{int} \mid P_1(x,\nu)\})$ and $\mathtt{copy2} : (x : \mathbf{int} \to \{\nu : \mathbf{int} \mid P_2(x,\nu)\})$.

$$\begin{array}{lcl} \mathtt{copy1} & : & (x : \mathbf{int} \to \{\nu : \mathbf{int} \mid P_1(x,\nu)\}) \\ \mathtt{copy2} & : & (x : \mathbf{int} \to \{\nu : \mathbf{int} \mid P_2(x,\nu)\}) \end{array}$$

By using them, we obtain the following set $C_{\mathtt{copy}}$ of constraints:

$$\begin{array}{rcl} x = 0 \wedge y = 0 & \Rightarrow & P_2(x,y) \\ P_2(x-1,y) \wedge x \neq 0 \wedge z = 1 + y & \Rightarrow & P_1(x,z) \\ P_1(n,x) & \Rightarrow & x = n \end{array}$$

4.1.2 Constraint Solving

Given a set C of non-recursive Horn clauses, our previous constraint solving algorithm returns a substitution θ for predicate variables in C such that θC is valid. The algorithm iteratively finds a solution for each predicate variable P in C as follows: The algorithm first computes equi-satisfiable constraints C_P of the following form by eliminating the other predicate variables in C than P:

$$\phi_P \Rightarrow P(\widetilde{x}) \qquad P(\widetilde{x}) \Rightarrow \phi'_P$$

Here, $FV(\phi_P) \cap FV(\phi'_P) \subseteq \{\widetilde{x}\}$ always holds. Intuitively, the predicate $P(\widetilde{x})$ represents an invariant of some subexpression e in the SHP, where some variable $\nu \in \{\widetilde{x}\}$ represents the value of e and each variable in $\{\widetilde{x}\} \setminus \{\nu\}$ represents a free variable in e. ϕ_P and ϕ'_P respectively represent the strongest condition satisfied by the value ν and the weakest condition on ν required by the context of e. The algorithm then computes $\mathcal{I}(\phi_P, \neg\phi'_P)$ as a solution for $P(\widetilde{x})$ with the help of a technique called interpolation [4, 16] from automated theorem proving. Here, an interpolant $\mathcal{I}(\phi_1, \phi_2)$ of ϕ_1 and ϕ_2 (such that ϕ_1 and ϕ_2 are inconsistent) is a formula ϕ that satisfies the following conditions:[4]

- ϕ_1 implies ϕ,
- ϕ and ϕ_2 are inconsistent, and
- $FV(\phi) \subseteq FV(\phi_1) \cap FV(\phi_2)$.

For the running example $C_{\mathtt{copy}}$, we obtain the following constraints by eliminating the other predicate variables than P_1:

$$\begin{array}{rcl} x = 1 \wedge y = 0 \wedge x \neq 0 \wedge \nu = 1 + y & \Rightarrow & P_1(x,\nu) \\ P_1(x,\nu) & \Rightarrow & \nu = x \end{array}$$

[3] For the sake of simplicity, we here omit the type template of main as well as the refinement predicates for the argument of copy1 and copy2.

[4] Note that interpolants of ϕ_1 and ϕ_2 are not unique. Actually, existing theorem provers [4, 16] return one of them, which is denoted by $\mathcal{I}(\phi_1, \phi_2)$.

We then obtain, for example, the following solution for $P_1(x,\nu)$:

$$\mathcal{I}(x = 1 \wedge y = 0 \wedge x \neq 0 \wedge \nu = 1 + y, \neg\nu = x) \equiv \nu = x.$$

By substituting this for P_1 in $C_{\mathtt{copy}}$, we get:

$$\begin{array}{rcl} x = 0 \wedge \nu = 0 & \Rightarrow & P_2(x,\nu) \\ P_2(x,\nu) & \Rightarrow & (x + 1 \neq 0 \wedge z = 1 + \nu \Rightarrow z = x + 1) \end{array}$$

We then get, for example, the following solution for $P_2(x,\nu)$:

$$\begin{array}{rl} & \mathcal{I}(x = 0 \wedge \nu = 0, \neg(x + 1 \neq 0 \wedge z = 1 + \nu \Rightarrow z = x + 1)) \\ \equiv & \nu = x. \end{array}$$

We thus obtain the following refinement types for $D_{\mathtt{copy}}$:

$$\begin{array}{lcl} \mathtt{copy1} & : & (x : \mathbf{int} \to \{\nu : \mathbf{int} \mid x = \nu\}) \\ \mathtt{copy2} & : & (x : \mathbf{int} \to \{\nu : \mathbf{int} \mid x = \nu\}) \end{array}$$

4.2 Limitation of Previous Method

We now explain the limitation of the previous method by using the program in Figure 4. Let us consider the following SHP $D_{\mathtt{cc}}$:

```
let copy1 x = assume (x<>0); 1 + copy2 (x-1)
let copy2 x = assume (x=0); 0
let copy3 x = assume (x<>0); 1 + copy4 (x-1)
let copy4 x = assume (x=0); 0
let main n = assume (copy3 (copy1 n) <> n); fail
```

The SHP corresponds to an infeasible error path where the else-branch of `copy` is taken in the first and the third calls of `copy`, the then-branch is taken in the second and the fourth calls of `copy`, and the assertion in the `main` function fails.

For the SHP $D_{\mathtt{cc}}$, we use the following type template: $x : \mathbf{int} \to \{\nu : \mathbf{int} \mid P_i(x,\nu)\}$ for each `copy`i. We get the following set $C_{\mathtt{cc}}$ of constraints:

$$\begin{array}{rcl} x = 0 \wedge y = 0 & \Rightarrow & P_2(x,y) \\ P_2(x-1,y) \wedge x \neq 0 \wedge z = 1 + y & \Rightarrow & P_1(x,z) \\ x = 0 \wedge y = 0 & \Rightarrow & P_4(x,y) \\ P_4(x-1,y) \wedge x \neq 0 \wedge z = 1 + y & \Rightarrow & P_3(x,z) \\ P_1(n,x) \wedge P_3(x,y) & \Rightarrow & y = n \end{array}$$

By eliminating the other predicate variables than P_3 (and with some simplification), we get the following constraints C_{P_3}:

$$\begin{array}{rcl} x = 1 \wedge \nu = 1 & \Rightarrow & P_3(x,\nu) \\ P_3(x,\nu) & \Rightarrow & (x = 1 \Rightarrow \nu = 1) \end{array}$$

Existing interpolating provers such as [4] returns the following solution for $P_3(x,\nu)$:

$$\mathcal{I}(x = 1 \wedge \nu = 1, \neg(x = 1 \Rightarrow \nu = 1)) \equiv x = 1 \wedge \nu = 1.$$

Note here that the solution is specific to the calling context of the particular function `copy3`, and cannot be used as a solution for P_2 and P_4. We here want to get more general solutions like $\lambda(\nu,x).\nu = x$ which are more likely to constitute an invariant of the function `copy` in the original program. For this purpose, we believe it is desirable to find the same solution (if possible) for "related" predicate variables which represent (possibly different) refinement predicates for the same argument or return value of the same function in the original program. For the running example $C_{\mathtt{cc}}$, we want to get the same solution for $P_1,\ldots,P_4$, and $\lambda(\nu,x).\nu = x$ in fact satisfies this extra constraint.

4.3 Extended Method

We now explain our extension of the previous method to remedy the limitation discussed in Section 4.2. The extended predicate discovery method is based on the framework of the previous method overviewed in Section 4.1, but the component for refinement type inference is extended so that it can merge and generalize information from multiple calling contexts of a function in multiple infeasible error paths. This enables MoCHi to infer a general refinement type of the function that type-checks the multiple calling contexts, while preserving the path- and context-sensitivity. In other words, the extended method generates constraints from multiple infeasible error paths (see Section 4.3.1), and tries to find the same solution (if possible) for related predicate variables (see Section 4.3.2).

4.3.1 Extensions of Constraint Generation

We extend the previous constraint generation algorithm overviewed in Section 4.1.1 as follows.

• For each CEGAR iteration, we generate constraints from multiple infeasible error paths instead of a single path: We keep the set $\{\pi_1,\cdots,\pi_n\}$ of the infeasible error paths found so far, generate the set C_i of Horn clauses for each path π_i, and pass $C = C_1 \cup \cdots \cup C_n$ to the extended constraint solving algorithm described in Section 4.3.2 as an input.

• We also construct and pass an equivalence relation E on the predicate variables in C such that $P\ E\ Q$ if and only if the predicate variables P and Q represent (possibly different) refinement predicates for the same argument or return value of the same function in the original program. For example, we obtain the trivial equivalence relation $E_{\mathtt{cc}} = \{P_1,\ldots,P_4\} \times \{P_1,\ldots,P_4\}$ for $C_{\mathtt{cc}}$. The constraint solving algorithm in Section 4.3.2 exploits E to find general solutions for C.

Thus, the extended algorithm generates a pair (C,E) of Horn clauses C for multiple paths and an equivalence relation E on the predicate variables in C unlike the previous algorithm which generates only Horn clauses for a single path. Here, the pair (C,E) of constraints can be viewed as hierarchical constraints where C must be always satisfied and E should be satisfied if possible.

4.3.2 Extensions of Constraint Solving

In this section, we extend the previous constraint solving algorithm overviewed in Section 4.1.2. Given a pair (C,E) of Horn clauses C and an equivalence relation E on the predicate variables in C, the algorithm returns a substitution θ for the predicate variables in C such that θC is valid. A distinguishing feature of the algorithm is that it tries to find the same solution for predicate variables related by E if possible. This enables the algorithm to obtain general predicates, which are more likely to constitute invariants.

The extended constraint solving algorithm proceeds as follows:

1. Find a set S of predicate variables which are related by E and may have the same solution in C.

2. Find a candidate solution $\lambda\widetilde{x}.\phi$ for all predicate variable $Q \in S$.

3. Substitute $\lambda\widetilde{x}.\phi$ for predicate variables S in C and repeat the entire procedure if the result still contains a predicate variable.

Finding a set S of predicate variables: To find a set of predicate variables that may have the same solution in C, for each predicate variable P in C, we compute constraints C_P from C by eliminating the other predicate variables than P. For the running example $C_{\mathtt{cc}}$, we obtain:

$$\begin{array}{rcl} C_{P_1} & = & \{x = 1 \wedge \nu = 1 \Rightarrow P_1(x,\nu), \\ & & \quad P_1(x,\nu) \Rightarrow (\nu = 1 \Rightarrow x = 1)\}, \\ C_{P_2} & = & \{x = 0 \wedge \nu = 0 \Rightarrow P_2(x,\nu), \\ & & \quad P_2(x,\nu) \Rightarrow (\nu = 0 \Rightarrow (x = -1 \vee x = 0))\}, \\ C_{P_3} & = & \{x = 1 \wedge \nu = 1 \Rightarrow P_3(x,\nu), \\ & & \quad P_3(x,\nu) \Rightarrow (x = 1 \Rightarrow \nu = 1)\}, \\ C_{P_4} & = & \{x = 0 \wedge \nu = 0 \Rightarrow P_4(x,\nu), \\ & & \quad P_4(x,\nu) \Rightarrow (x = 0 \Rightarrow \nu = 0)\}. \end{array}$$

Let $\{P_1,\ldots,P_m\}$ be the set of predicate variables in C. We pick an equivalence class $S_0 \in \{P_1,\ldots,P_m\}/E$ (e.g., the largest one), and further classify S_0 by using $C_{P_1},\ldots,C_{P_m}$ so that predicate variables which never have the same solution are separated. Formally, we find $S_1 \ldots, S_n$ such that:

- $S_0 = S_1 \cup \cdots \cup S_n$,
- $\phi_{P_{i,1}} \vee \cdots \vee \phi_{P_{i,\ell_i}}$ implies $\phi'_{P_{i,1}} \wedge \cdots \wedge \phi'_{P_{i,\ell_i}}$ for each $i \in \{1, \dots, n\}$, and
- $\phi_{P_{i,1}} \vee \cdots \vee \phi_{P_{i,\ell_i}} \vee \phi_{P_{j,1}} \vee \cdots \vee \phi_{P_{j,\ell_j}}$ does not imply $\phi'_{P_{i,1}} \wedge \cdots \wedge \phi'_{P_{i,\ell_i}} \wedge \phi'_{P_{j,1}} \wedge \cdots \wedge \phi'_{P_{j,\ell_j}}$ for each $i, j \in \{1, \dots, n\}$ such that $i \neq j$.

Here, $C_P = \{\phi_P \Rightarrow P(\widetilde{x}), P(\widetilde{x}) \Rightarrow \phi'_P\}$ and $S_i = \{P_{i,1}, \dots, P_{i,\ell_i}\}$.

We then pick some $S \in \{S_1, \dots, S_n\}$ (e.g., the largest one). For the running example $C_{\mathtt{cc}}$, we get $S = S_0 = S_1 = \{P_1, \dots, P_4\}$ since $\phi_{P_1} \vee \cdots \vee \phi_{P_4}$ implies $\phi'_{P_1} \wedge \cdots \wedge \phi'_{P_4}$.

Finding a candidate solution $\lambda\widetilde{x}.\phi$ ***for*** S***:*** We find a single candidate solution $\lambda\widetilde{x}.\phi$ for all the predicate variables $Q_1, \dots, Q_\ell \in S$ by simultaneously solving $C_{Q_1}, \dots, C_{Q_\ell}$ unlike the previous method. Formally, we find ϕ such that:

- $\phi_{Q_1} \vee \cdots \vee \phi_{Q_\ell}$ implies ϕ,
- ϕ implies $\phi'_{Q_1} \wedge \cdots \wedge \phi'_{Q_\ell}$, and
- $FV(\phi) \subseteq \{\widetilde{x}\}$.

Here, $C_{Q_i} = \{\phi_{Q_i} \Rightarrow Q_i(\widetilde{x}), Q_i(\widetilde{x}) \Rightarrow \phi'_{Q_i}\}$. We can compute such a formula ϕ as an interpolant $\mathcal{I}(\phi_{Q_1} \vee \cdots \vee \phi_{Q_\ell}, \neg(\phi'_{Q_1} \wedge \cdots \wedge \phi'_{Q_\ell}))$ but the three conditions of interpolants are not always sufficient for our purpose to find general predicates. Actually, we want to obtain as simple interpolant as possible with respect to the number of disjunctions. To this end, we propose a new heuristic operator $\mathcal{J}$ that combines the interpolation $\mathcal{I}$ and convex hull operators. Let us write $\mathcal{H}(\phi)$ to denote the convex hull of ϕ. For formulas ϕ_1 and ϕ_2 (such that ϕ_1 and ϕ_2 are inconsistent), the new operator $\mathcal{J}(\phi_1, \phi_2)$ is defined as follows:

$$\mathcal{J}(\phi_1, \phi_2) = \begin{cases} \mathcal{I}(\mathcal{H}(\phi_1), \mathcal{H}(\phi_2)) & (\text{if } \mathcal{H}(\phi_1) \perp \mathcal{H}(\phi_2)) \\ \mathcal{I}(\mathcal{H}(\phi_1), \phi_2) & (\text{if } \neg(\mathcal{H}(\phi_1) \perp \mathcal{H}(\phi_2)) \wedge \\ & \quad \mathcal{H}(\phi_1) \perp \phi_2) \\ \mathcal{I}(\phi_1, \phi_2) & (\text{otherwise}) \end{cases}$$

Here, we write $\phi_1 \perp \phi_2$ to denote that ϕ_1 and ϕ_2 are inconsistent. Note here that the use of the convex hull operator enables us to eliminate disjunctions in ϕ_1 and ϕ_2, which are passed to an interpolating theorem prover. In the experiments reported in Section 6, this often reduced the number of disjunctions in the output of the interpolating prover, and hence makes the output more likely to constitute invariants. Thus, we use the new operator $\mathcal{J}$ instead of $\mathcal{I}$ to compute $\lambda\widetilde{x}.\mathcal{J}(\phi_{Q_1} \vee \cdots \vee \phi_{Q_\ell}, \neg(\phi'_{Q_1} \wedge \cdots \wedge \phi'_{Q_\ell}))$ as a candidate solution $\lambda\widetilde{x}.\phi$ for all the predicate variables $Q_1, \dots, Q_\ell \in S$. For the running example $C_{\mathtt{cc}}$, we obtain, for example, the following candidate solution $\lambda(x, \nu).\phi$ for $P_1, \dots, P_4$:

$$\begin{array}{rcl} \phi & = & \mathcal{J}(\phi_{P_1} \vee \cdots \vee \phi_{P_4}, \neg(\phi'_{P_1} \wedge \cdots \wedge \phi'_{P_4})) \\ & = & \mathcal{I}(\mathcal{H}(x = \nu = 0 \vee x = \nu = 1), \neg(\phi'_{P_1} \wedge \cdots \wedge \phi'_{P_4})) \\ & = & \mathcal{I}(0 \leq x = \nu \leq 1, \neg(\phi'_{P_1} \wedge \cdots \wedge \phi'_{P_4})) \\ & \equiv & x = \nu \end{array}$$

Substituting $\lambda\widetilde{x}.\phi$ ***for*** S ***in*** C***:*** We then substitute the candidate solution $\lambda\widetilde{x}.\phi$ for $S = \{Q_1, \dots, Q_\ell\}$ in C. Note, however, that we cannot always substitute all the predicate variables in S with the candidate solution $\lambda\widetilde{x}.\phi$ because Q_i may depend on Q_j for some $i \neq j$. For example, let us consider the following constraints:

$$x = 0 \Rightarrow Q_1(x), \quad Q_1(x) \Rightarrow Q_2(x+1), \quad Q_2(x) \Rightarrow 0 \leq x \leq 2$$

From the constraints, we get:

$$\begin{array}{rcl} C_{Q_1} & = & \{\nu = 0 \Rightarrow Q_1(\nu), Q_1(\nu) \Rightarrow -1 \leq \nu \leq 1\}, \\ C_{Q_2} & = & \{\nu = 1 \Rightarrow Q_2(\nu), Q_2(\nu) \Rightarrow 0 \leq \nu \leq 2\}. \end{array}$$

Thus, we obtain, for example, $\mathcal{J}(\nu = 0 \vee \nu = 1, \neg(-1 \leq \nu \leq 1 \wedge 0 \leq \nu \leq 2)) \equiv 0 \leq \nu \leq 1$ as a candidate solution for $Q_1(\nu)$ and $Q_2(\nu)$. However, $[\lambda\nu.0 \leq \nu \leq 1/Q_1, \lambda\nu.0 \leq \nu \leq 1/Q_2](Q_1(x) \Rightarrow Q_2(x+1))$ is not valid. Actually, it is only safe to substitute $\lambda\nu.0 \leq \nu \leq 1$ for either Q_1 or Q_2.

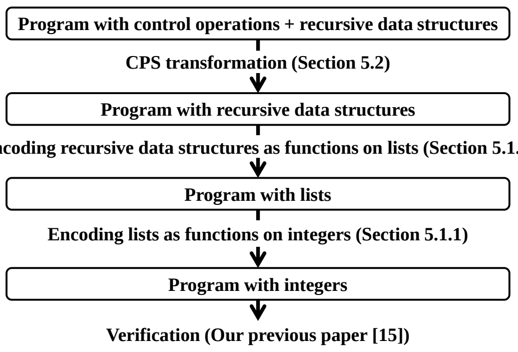
Program with control operations + recursive data structures
CPS transformation (Section 5.2)
Program with recursive data structures
Encoding recursive data structures as functions on lists (Section 5.1.2)
Program with lists
Encoding lists as functions on integers (Section 5.1.1)
Program with integers
Verification (Our previous paper [15])

Figure 10. The Verification Framework for Recursive Data Structures and Control Operators

Therefore, we find and substitute only a maximal nonempty subset M of S for which we can safely substitute $\lambda\widetilde{x}.\phi$ (i.e., $\{R \mapsto \lambda\widetilde{x}.\phi \mid R \in M\}C$ is equi-satisfiable with C). For the running example $C_{\mathtt{cc}}$, it is in fact safe to substitute the candidate solution $\lambda(x, \nu).x = \nu$ for $P_1, \dots, P_4$ (i.e., $M = S$). As a result of the substitution, all the predicate variables in $C_{\mathtt{cc}}$ are eliminated. Thus, we obtain the refinement type $x : \textbf{int} \to \{\nu : \textbf{int} \mid x = \nu\}$ for all of $\mathtt{copy1}, \dots, \mathtt{copy4}$.

5. Language Extensions

This section formalizes extensions of the target language of verification. Our approach is to translate a source program to a program that has no recursive data structures and no control operators, in a sound and complete manner. Figure 10 shows the verification framework for recursive data structures and control operators. Section 5.1 formalizes the encoding of recursive data structures, and Section 5.2 introduces the extension for control operations.

5.1 Functional Encoding of Recursive Data Structures

We first discuss encoding of lists, and then that of user-defined recursive data structures. We assume that the target language of encoding is equipped with tuples. The extensions of selective predicate abstractions and selective CPS transformation with tuples are straightforward.

5.1.1 Functional Encoding of Lists

The idea is to encode a list into a pair of its length and a function that maps indices to the elements of the list. For example, the list $[2; 3; 5]$ is encoded into the pair $(3, f)$ where $f(0) = 2$, $f(1) = 3$, and $f(2) = 5$. The primitive operations `nil`, `cons`, `is_nil`, `head`, and `tail` for lists are defined as follows.

```
let nil = (0, fun _ -> fail)
let cons x (len,f) = (len+1,
  λi. if i = 0 then x else f (i-1))
let is_nil (len,f) = len = 0
let head (len,f) = if len=0 then fail else f 0
let tail (len,f) =
  ((if len=0 then fail else len-1), λi.f(i+1))
```

`nil` is translated into the pair of length 0 and the function that always fails. `cons x xs` is translated into the pair of its length and the function $\{0 \mapsto \mathtt{x}\} \cup \{i \mapsto f(i-1) \mid i \neq 0\}$ where f is the function part of the encoding of `xs`. `is_nil` just checks whether `len` is 0 or not. `head` returns `f(0)`, i.e. the first element of the list. `tail` returns the pair of `(len-1,f')` where `f'(i) = f(i+1)`.

Note that we cannot use Church encoding for recursive data structures, since Church encoding of data structures require recursive or polymorphic types in general, which cannot be handled by higher-order model checking.

Our approach has the following advantages. First, by encoding lists into functions over integers, we can reuse the predicate abstraction/discovery for integers. Second, the encoding induces a natural predicate abstraction of lists, which is general enough to subsume various abstractions known in the literature, such as Dillig et al.'s container abstraction [7]. With their abstraction method, a list is represented as $\{(v_1, P_1), \ldots, (v_n, P_n)\}$, which means the j-th element is v_i if $P_i(j)$ holds. For example, $\{(0, \mathit{true})\}$ denotes that all the elements are 0 and $\{(1, \lambda i.\, i \bmod 2 = 0)\}$ denotes that the even indexed elements are 1. By using our approach, the same information can be represented as a refinement type $(i : \mathbf{int}) \to \{x : \mathbf{int} \mid (P_1(i) \to x = v_1) \wedge \cdots \wedge (P_n(i) \to x = v_n)\}$. For example, $\{(1, \lambda i.\, i \bmod 2 = 0)\}$ is represented as $(i : \mathbf{int}) \to \{x : \mathbf{int} \mid i \bmod 2 = 0 \to x = 1\}$. Moreover, our approach can deal with list properties like "the i-th element of a list is greater than i," which cannot be represented by the container abstraction. Thus, our method is strictly more expressive.

5.1.2 Extension for Recursive Data Structures

We now discuss encoding of other recursive data structures. Programs with recursive data structures are translated into programs with lists by encoding recursive data structures to functions which map paths of nodes to labels. Here, a path and a label are represented as a list of integers and an integer respectively. For example, consider binary trees defined as follows.

```
type btree = Leaf | Node of btree * btree
```

A binary tree is encoded into a term of the type $\mathbf{int\ list} \to \mathbf{int}$.[5] For example, the tree $\mathbf{node}(\mathbf{leaf}, \mathbf{node}(\mathbf{leaf}, \mathbf{leaf}))$ is encoded into a function $\{[] \mapsto \mathbf{node}, [1] \mapsto \mathbf{leaf}, [2] \mapsto \mathbf{node}, [2, 1] \mapsto \mathbf{leaf}, [2, 2] \mapsto \mathbf{leaf}\}$ where $\mathbf{leaf}$ and $\mathbf{node}$ are defined as some integers. Here is another example.

```
match x with Constr1(x1, x2, ...) -> t
           | Constr2(...) -> ...
```

The expression above is encoded to the following expression.

```
let Constr1 = 1 in ... let Constrn = n in
  match x nil with
      Constr1 -> let x1 xs = x (cons 1 xs) in ...
                 let xn xs = x (cons n xs) in t'
    | Constr2 -> ...
```

Here, `t'` is the encoding of `t`. The pattern matching on trees is translated to that on labels, represented as integers. A subterm of a tree is obtained by adding the index to the head of the path.

For recursive data types, we impose the restriction that recursive type variables cannot occur under function constructors. Thus, $\mu\alpha.\, \mathbf{unit} + (\mathbf{int} \to \mathbf{int}) * \alpha$ (which corresponds to $(\mathbf{int} \to \mathbf{int})\ \mathbf{list}$) is OK, but neither $\mu\alpha.\, \alpha \to \mathbf{int}$ nor $\mu\alpha.\, (\mathbf{int} \to \alpha)$ is allowed.

Let $[\![-]\!]$ be the encoding discussed above. The transformed program is reduced to the same value as the original program:

Proposition 5.1 (Correctness of encoding). *Let t be a term in a program D. $t \longrightarrow_D^* \mathbf{fail}$ if and only if $[\![t]\!] \longrightarrow_D^* \mathbf{fail}$.*

5.2 Extension for Control Operations

We can extend the framework to deal with control operations (e.g., exceptions and `call/cc`) by removing them from a program by CPS transformation [17]. We do not support exceptions which carry function arguments, since the encoding of function-carrying exceptions by CPS requires recursive types.

The following program calculates factorial and raises an exception if a negative number or zero is given.

[5] Terms with this type is encoded to terms with type $(\mathbf{int} \to \mathbf{int}) \to \mathbf{int}$ by the list encoding.

```
exception NotPos
letrec fact n = if n <= 0 then raise NotPos
  else try n * fact (n - 1) with NotPos -> 1
let main n =
  try fact n with NotPos -> assert (n <= 0); 0
```

We can translate this program to an exception-free program by CPS transformation as follows:

```
letrec fact n k exn =
  if n <= 0 then exn NotPos
  else let exn' e = match e with NotPos -> k 1 in
         fact (n - 1) (fun r -> k (n * r))) exn'
let main n k = fact n k (fun e ->
  match e with NotPos -> assert (n <= 0); k 0)
```

Once the exception is removed, we can apply our verification method to the program.

6. Implementation and Preliminary Experiments

To evaluate the extended framework, we have implemented a prototype verifier for higher-order programs with lists and exceptions. Our verifier uses TRecS [12, 13] as the underlying higher-order model checker (for Step 3 in Figure 1), and uses CSIsat [4] for predicate discovery (in Step 5). CVC3 [2] is used for unsafety check (in Step 4) and predicate abstraction (in Step 1).

Table 1 shows the results of the experiments. The column "size" shows the word counts of the program. The last column shows the number of CEGAR-cycles and the running time measured. In the last column, "C.", "A.", and "D." denote the uses of selective CPS transformation, selective predicate abstraction, and refined predicate discovery, respectively. The programs have been verified correctly. The experiment was conducted on Intel Xeon 5570 CPU with 8 MB cache and 6 GB memory. The implementation can be tested and all programs are available at `http://www.kb.ecei.tohoku.ac.jp/~ryosuke/mochi/`.

The programs used in the experiments are:

- "r-file" and above are the programs used in the experiments in the previous paper [15].
- "sum_intro", "copy_intro", and "fact_notpos" are the example programs in Section 1 and Section 5.2.
- "map_filter" and "risers" are examples of Ong and Ramsay's verification framework [19] for higher-order recursion scheme with a *case* construct listed at their web page `http://mjolnir.cs.ox.ac.uk/cgi-bin/horsc/recheck-horsc/input`. Our framework can verify these programs without a special treatment of case constructs unlike in their framework.
- "search" is a program that manipulates user-defined data structures.
- Other programs define generators of lists and functions on lists, and assert that the functions work correctly. For example, "zip" defines a function that takes two lists and returns a list of corresponding pairs. The function fails if the two arguments have different lengths. "zip" asserts that `zip xs xs` never fails for all integer lists `xs`.
- A program of name "xxx-e" is a buggy version of the program "xxx".

The selective CPS transformation and the selective predicate abstraction reduced the time required for verification, and enabled verification of various programs including "a-cppr", "zip", and "map_head_filter", which could only be verified by using both of them. Especially, the selective CPS transformation reduced the time required for higher-order model checking and the selective predicate abstraction reduced the number of CEGAR cycles as expected (recall their advantages discussed in Section 1). The refined predicate discovery (especially in combination with the selective predicate abstraction) enabled verification of not only "copy_intro" but also the list-manipulating programs "length", "nth", "risers", and

Table 1. Results of preliminary experiments

			cycle, time [sec]							
program	size	order	none	C.	A.	C. & A.	D.	C. & D.	A. & D.	C. & A. & D.
sum	24	1	2, 0.12	2, 0.12	1, 0.07	1, 0.08	2, 0.12	2, 0.11	1, 0.07	1, 0.07
mult	**31**	**1**	-	**4, 53.67**	**3, 0.14**	**2, 0.13**	**4, 0.78**	**4, 0.61**	**3, 0.18**	**3, 0.17**
max	42	2	5, 4.32	5, 1.02	1, 0.20	0, 0.08	5, 17.10	5, 1.18	3, 1.24	0, 0.08
mc91	32	1	2, 0.19	2, 0.20	2, 0.18	2, 0.18	2, 0.32	2, 0.32	2, 0.40	2, 0.39
ack	**53**	**1**	-	-	**1, 0.11**	**1, 0.10**	**4, 1.08**	**5, 0.50**	**1, 0.12**	**1, 0.10**
a-cppr	**149**	**2**	-	-	-	**7, 4.41**	-	-	-	**7, 2.03**
l-zipunzip	**81**	**2**	-	**5, 3.15**	-	**2, 0.14**	-	**4, 0.60**	**5, 1.88**	**2, 0.13**
l-zipmap	65	2	7, 1.36	6, 0.44	2, 0.14	2, 0.11	7, 1.36	6, 0.59	3, 0.19	4, 0.23
hors	64	2	2, 0.43	2, 0.11	1, 0.16	1, 0.07	2, 3.91	2, 0.11	1, 0.16	1, 0.07
e-simple	27	2	1, 0.08	1, 0.07	0, 0.06	0, 0.06	1, 0.08	1, 0.07	0, 0.06	0, 0.06
e-fact	55	2	2, 0.13	2, 0.10	2, 0.09	2, 0.10	2, 0.15	2, 0.11	2, 0.11	2, 0.10
r-lock	54	1	6, 0.83	6, 0.39	0, 0.08	0, 0.07	6, 0.94	6, 0.39	0, 0.07	0, 0.08
r-file	**168**	**1**	-	**17, 27.87**	**10, 2.69**	**7, 0.92**	-	-	**8, 3.19**	**6, 1.50**
sum_intro	33	1	2, 0.17	2, 0.14	1, 0.07	1, 0.08	2, 0.18	2, 0.14	1, 0.07	1, 0.08
copy_intro	**24**	**1**	-	-	-	-	**3, 0.36**	**3, 0.35**	**2, 0.14**	**2, 0.14**
fact_notpos	97	1	3, 0.28	3, 0.25	2, 0.11	2, 0.11	3, 0.51	3, 0.49	2, 0.14	2, 0.12
fold_right	**64**	**2**	-	**8, 84.61**	**2, 0.45**	**2, 0.22**	-	-	**2, 1.04**	**2, 0.31**
forall_eq_pair	**55**	**1**	-	-	**2, 0.38**	**1, 0.20**	-	-	**2, 0.39**	**1, 0.22**
forall_leq	**55**	**2**	**6, 17.54**	-	**2, 0.36**	**1, 0.19**	-	-	**2, 0.33**	**1, 0.22**
isnil	52	1	3, 0.22	3, 0.20	2, 0.13	2, 0.12	3, 0.35	3, 0.22	2, 0.12	2, 0.12
iter	**59**	**2**	-	**7, 46.22**	**1, 0.18**	**1, 0.16**	-	-	**1, 0.21**	**1, 0.18**
length	**49**	**1**	-	-	-	-	**4, 1.42**	**2, 0.24**	**2, 0.14**	**2, 0.14**
mem	74	1	5, 3.46	4, 0.70	3, 0.46	3, 0.30	-	7, 17.83	4, 0.61	4, 0.37
nth	**59**	**1**	-	-	-	-	-	**3, 1.18**	**4, 0.55**	**4, 0.42**
nth0	78	1	3, 0.48	3, 0.30	3, 0.26	3, 0.22	4, 3.77	4, 1.00	3, 0.28	3, 0.21
harmonic	**101**	**2**	-	-	**1, 0.39**	**1, 0.20**	-	-	**1, 0.72**	**1, 0.25**
fold_left	**64**	**2**	-	-	**2, 0.39**	**2, 0.22**	-	-	**2, 0.79**	**2, 0.30**
zip	**69**	**1**	-	-	-	**6, 24.16**	-	**7, 32.45**	**7, 5.23**	**8, 22.17**
map_filter	**111**	**2**	-	-	-	**3, 39.84**	-	-	**4, 5.46**	**3, 5.82**
risers	**79**	**1**	-	-	-	-	-	-	**8, 18.67**	**8, 9.78**
search	**109**	**2**	-	**7, 4.20**	**4, 2.28**	**3, 0.71**	-	**8, 15.49**	**5, 4.65**	**8, 9.65**
fold_fun_list	**78**	**3**	-	-	-	-	-	-	**2, 21.40**	**2, 2.74**
fact_notpos-e	97	1	1, 0.12	1, 0.11	1, 0.09	1, 0.09	1, 0.12	1, 0.14	1, 0.09	1, 0.08
harmonic-e	101	2	0, 0.11	1, 0.16	0, 0.09	0, 0.08	0, 0.11	1, 0.19	0, 0.10	0, 0.08
map_filter-e	**111**	**2**	**4, 8.10**	**2, 0.83**	**3, 2.16**	**0, 0.13**	-	**2, 5.62**	**5, 7.65**	**0, 0.13**
search-e	**78**	**2**	**5, 12.56**	**4, 0.91**	**2, 0.65**	**1, 0.18**	-	**5, 6.00**	**2, 0.85**	**4, 1.50**

"fold_fun_list" by finding general predicates for abstraction as discussed in Section 1. Note here that "fold_fun_list" is order-3 and the other order-1 list-manipulating programs were transformed to order-2 programs before verification by encoding lists as functions.

It is also worth noting that the columns "... & D." show the experimental results for the refined predicate discovery without the feature of merging multiple infeasible paths enabled. The feature actually slowed down verification of some programs (such as "map_filter") but improved the analysis precision of MoCHi. In particular, the feature enabled us to verify the following accumulator version of "length" in 0.36 seconds with 3 CEGAR-cycles:

```
let rec length acc xs = match xs with
  [] -> acc | _::xs' -> length (acc+1) xs'
let rec make_list n =
  if n = 0 then [] else n :: make_list (n-1)
let main n = assert (length 0 (make_list n) = n)
```

For the success of verification, an abstraction predicate $\mathrm{length}(\mathtt{r}) = \mathrm{length}(\mathtt{acc}) + \mathrm{length}(\mathtt{xs})$ on the return value `r` and the arguments `acc` and `xs` of `length` was essential. The predicate could only be found if the feature of merging multiple paths was enabled.

7. Related Work

7.1 Verification of Higher-Order Programs with Recursive Data Structures

Ong and Ramsay [19] proposed a verification method for functional programs with recursive data structures, called Pattern Matching Recursion Schemes (PMRS). The method cannot handle regular properties (such as "a and b occur alternately") and numerical properties (such as "$x + y \leq z$" where x, y, z are the length of lists).

Unno et al. [25] also proposed a verification method for higher-order tree processing functional programs, which is based on a verification method for higher-order multi-tree transducers [14]. Their method can verify regular properties of recursive data structures provided that certain invariant annotations are given.

There are several studies [5, 10, 11, 21, 24–28] that aim to infer dependent types for higher-order programs with recursive data structures. Rondon et al.'s liquid type inference [11, 21] is a semi-automated verification method that requires users to provide templates of predicates, called logical qualifiers. The expressive power of their method and ours is incomparable. They can deal with "recursive dependent types", such as $\mathbf{int\ list}_{\leq} = \mu t.\, \mathbf{nil} + \mathbf{cons}(x_1 : \mathbf{int}, \{\nu : \mathbf{int} \mid x_1 \leq \nu\}\ t)$, which represents ordered lists of integers, while our method cannot. On the other hand, our method can deal with the properties of list elements related to their indices like "the i-th element of a list is greater than i," while they cannot. Unno and Kobayashi [24], and Jhala et al. [10] proposed a method for automated refinement type inference, where templates of refinement types are first prepared and then constraints on unknown refinement predicates are generated. Unno and Kobayashi [24] then solve the constraints by using an interpolating theorem prover, and Jhala et al. [10] solves the constraints by a reduction to model-checking of first-order programs. Those methods can deal with data structures such as lists and arrays as long as type templates for the data structures are given a priori. Compared with our "data structures as functions" approach, however, the supported properties seem to be limited; for example, their method cannot reason about a relation between a list index and the corresponding element (like "the i-th element is greater than i"). Unlike our method, the dependent type inference methods stated above [10, 11, 21, 24] do not support refinement intersection types, which are necessary for precise, context-sensitive analysis of higher-order functions.

Xi and Pfenning [26] proposed a dependently-typed language Dependent ML. Its type system captures program properties such as absence of array bounds errors and violations of data structure invariants. Unlike our method, Dependent ML requires users to provide dependent types of top-level functions.

7.2 Automated Verification of First-Order Programs with Recursive Data Types

Chin et al. [5] proposed sized type inference. Their method infers invariant of recursive functions by fixed-point computation. By abstracting lists as multisets, their method can deal with the inclusion relation and the membership relation on lists. For example, their method can verify that `exists x xs` returns true if and only if `xs` has `x` as an element. The method, however, cannot properly handle higher-order functions.

Suter et al. [23] proposed a verification method for first-order functional programs that manipulate recursive data structures. They use a decision procedure [22] that is complete for recursive functions that are sufficiently surjective catamorphisms. For example, their method can verify that an insertion function preserves invariants of binary search trees, while our method cannot. On the other hand, their method cannot deal with higher-order functions in a context-sensitive way unlike our method.

Dillig et al. [7] proposed an automatic technique for statically reasoning about containers. The proposed method is based on an abstract interpretation for containers. Their method is similar to our method in the sense that they model containers as mappings from locations to values. They consider only a client-side use of specific data structures (i.e. containers) such as primitive data structures and those defined in a standard library. In contrast, our method can deal with user-defined data structures. Moreover, as discussed in Section 5, our method is strictly more expressive than their method.

7.3 Path-sensitive Predicate Discovery

Beyer et al. [3] proposed a predicate discovery method for imperative programs. Like our method, theirs addresses the problem of finding general predicates in the context of a path-sensitive analysis. The method uses full-fledged programs (called path programs) as counterexamples instead of infeasible error paths (or straightline programs). A path program can be viewed as a set of finite paths that are obtained by unwinding the path program. By synthesizing invariants of a path program, their method prevents appearance of infinite simple variations of infeasible error paths. Our method, in a sense, generalizes theirs to path- and context-sensitive verification of higher-order programs.

8. Conclusion

We have proposed extensions and refinements to realize a scalable software model checker for higher-order programs. We have identified the problems of the previous verification method, and proposed the optimization techniques to overcome the problems. We have implemented a prototype verifier for higher-order programs with lists, which works well for several programs.

Acknowledgment

We would like to thank anonymous referees for useful comments. This work was partially supported by Kakenhi 24·8057 and 23220001.

References

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

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

[3] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI 2007*, pages 300–309, June 2007.

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

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

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

[7] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL 2011*, pages 187–200, 2011.

[8] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 1997*, pages 72–83, 1997.

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

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

[11] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI 2009*, pages 304–315, 2009.

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

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

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

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

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

[17] L. R. Nielsen. A selective CPS transformation. *Electronic Notes in Theoretical Computer Science*, 45:311–331, 2001.

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

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

[20] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *TCS*, 1(2):125–159, 1975.

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

[22] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL 2010*, pages 199–210, 2010.

[23] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS 2011*, pages 298–315, 2011.

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

[25] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing programs via higher-order model checking. In *APLAS 2010*, pages 312–327, 2010.

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

[27] D. N. Xu. Hybrid contract checking via symbolic simplification. In *PEPM 2012*, pages 107–116, Jan. 2012.

[28] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for haskell. In *POPL 2009*, pages 41–52, 2009.