Hybrid Contract Checking via Symbolic Simplification

Dana N. Xu
INRIA
na.xu@inria.fr

Abstract
Program errors are hard to detect or prove absent. Allowing programmers to write formal and precise specifications, especially in the form of contracts, is one popular approach to program verification and error discovery. We formalize and implement a hybrid contract checker for a subset of OCaml. The key technique we use is symbolic simplification, which makes integrating static and dynamic contract checking easy and effective. Our technique statically verifies that a function satisfies its contract or blames the function violating the contract. When a contract satisfaction is undecidable, it leaves residual code for dynamic contract checking.

Categories and Subject Descriptors D.3 [Software]: Programming Languages

General Terms symbolic simplification, functional language, verification, debugging

Keywords contract semantics, static, dynamic, hybrid, contract checking

1. Introduction
Constructing reliable software is difficult even with functional languages. Formulating and checking (statically or dynamically) logical assertions [2, 5, 16, 18, 37], especially in the form of contracts [7, 13, 14, 30, 41], is one popular approach to error discovery. Static contract checking can catch all contract violations but may give false alarm and can only check restricted properties; dynamic checking can check more expressive properties but consumes runtime cycles and only checks the actual executed paths, thus is not complete. Static and dynamic checking can be complementary. In this paper, we formalize hybrid (i.e. static followed by dynamic) contract checking for a subset of OCaml. Thus, no (potential) contract violations can escape and yet expressive properties can be expressed.

Consider an OCaml program augmented with a contract declaration:

\[
(* \text{val } f1 : \text{int} -> \text{int} -> \text{int} *)
contract f1 = \{(x \mid x >= 0) \rightarrow \{(y \mid y >= 0) \rightarrow \{z \mid z >= 0\}\}\}
let f1 g = (g 1) - 1
let f2 = f1 (fun x -> x - 1)
\]

The contract of \( f1 \) says that if \( f1 \) takes a function that returns a non-negative number when given a non-negative number, the function \( f1 \) itself returns a non-negative number. Both a static checker and a dynamic checker are able to report that \( f1 \) fails its precondition: the static checker relies on the invalidity of \( \forall g : \text{int} \rightarrow \text{int}. (g 1) \geq 0 \Rightarrow (g 1) - 1 \geq 0 \) while the dynamic checker evaluates \( (((\text{fun } x \rightarrow x - 1) 1) - 1) \rightarrow -1 \), which violates the contract \( \{z \mid z >= 0\} \). However, a dynamic checker cannot tell that the argument \( (\text{fun } x \rightarrow x - 1) \) fails \( f1 \)'s precondition because there is no witness at run-time, while a static checker can report this contract violation because \( x - 1 \geq 0 \) does not hold for all \( x \) of \text{int} to satisfy the postcondition \( \{y \mid y \geq 0\} \). On the other hand, a static checker usually gives three outcomes: (a) definitely no bug; (b) definitely a bug; (c) possibly a bug. Here, a bug refers to a contract violation. If we get many alarms (c), it may take us a lot of time to check which one is a real bug and which one is a false alarm. We may want to invoke a dynamic checker when the outcome is (c).

Following the formalization in [41], but this time for a strict language. We first give a denotational semantics to contract satisfaction. That is to define what it means by an expression \( e \) satisfies its contract \( t \) (written \( e \in t \)) without knowing its implementation. Next, we define a wrapper \( t_0 \) that takes an expression \( e \) and its contract \( t \) and produces a term \( e \in t_0 \) such that contract checks are inserted at appropriate places in \( e \). If a contract check is violated, a special constructor \text{BAD} signals the violation. As the term \( e \in t_0 \) is a term in the same language as \( e \), all we have to do is to check the reachability of \text{BAD}. If a \text{BAD} is reachable, we know a contract is violated and the label \( l \) precisely captures the function at fault. We symbolically simplify the term \( e \in t_0 \) aiming to simplify \text{BAD}s away. In case there is any \text{BAD} left, we either report it as a compile-time error or leave the residual code for dynamic checking. We make the following contributions:

- We clarify the relationship between static contract checking and dynamic contract checking (§2). A new observation is that, after static checking, we should prune away some more unreachable code before go on dynamic checking. Such unreachable code however is essential during static checking. We prove the correctness of this pruning (§6) with the telescoping property studied (but not used for such purpose) in [7, 41].
- We define \( e \in t \) and \( e \in t_0 \) and prove a theorem \( "e\in t_\text{BAD}& is crash-free \equiv e \in t_0" (§4)\). The "crash-free" means "\text{BAD} is not reachable under all contexts". Such a formalization is tricky and its correctness proof is non-trivial. We re-do the kind of proofs in [42] for a strict language.
- We design a novel SL machine that augments symbolic simplification with contextual information synthesis for checking the reachability of \text{BAD} statically (§3). The difficulty lies in the reasoning about non-total terms. The checking is automatic and modular and we prove is soundness. Moreover, the SL machine
produces residual code for dynamic checking. We compare our framework with other approaches in §7.

- We design a logization technique that transforms expressions to logical formulae, inspired by [19, 20] and axiomatization of functions that interactive theorem provers perform before calling SMT solvers. However, we have to deal with non-total terms and that is the key contribution of the logization (§5).

2. Overview

Assertions [18] state logical properties of an execution state at arbitrary points in a program; contracts specify agreements concerning the values that flow across a boundary between distinct parts of a program (modules, procedures, functions, classes). If an agreement is violated, contract checking is supposed to precisely blame the function at fault. Contracts were first introduced to be checked at run-time [13, 30]. To perform dynamic contract checking (DCC), a function must be called to be checked. For example:

```plaintext
contract inc = { x | x > 0 } -> { y | y > 0 }
let inc = fun v -> v + 1
let tl = inc 0
A dynamic checker wraps the inc in tl with its contract tinc:

let tinc = (inc ⊲ tlinc) 0

where l is (2, 5, “inc”) indicating the source location where inc is defined (row:col:5) and t is (3, 10, “tl1”) indicating the location of the call site with caller’s name. This wrapped tinc expands to:

(∀x. let y = inc (let x = x1 in if x > 0 then x else BAD(3,10,tl1))
                               in if y > 0 then y else BAD(2,5,“inc”)) 0

In the upper box, the argument of inc is guarded by the check x > 0; in the lower box, the result of inc is guarded by the check y > 0. If a check succeeds, the original term is returned; otherwise, the special constructor BAD is reached and a blame is raised. In this case, tinc has no blame, which fails inc’s precondition. Running the above wrapped code, we get BAD(3,10,tl1), which precisely blames tl.

The DCC algorithm is like this. Given a function f and a contract t, to check that the callee f and its caller agree on the contract t dynamically, a checker wraps each call to f with its contract:

f ⊲ t

which behaves the same as f except that (a) if f disobeys t, it blames f, signaled by BAD(f); (b) if the context uses f in a way not permitted by t, it blames the caller of f, signaled by BAD where “?” is filled with a caller name and the call site location.

Later, [7, 41] give formal declarative semantics for contract satisfaction that not only allow us to prove the correctness of DCC w.r.t. this semantics, but also to check contracts statically.

The essence of static contract checking (SCC) is:

splitting ⊲ into half: e ⊲ t = e ⊲ t and e ⊲ t = e ⊲ t.

The “ensures” and the “requires” are dual to each other. The special constructor UNR (pronounced “unreachable”), does not raise a blame, but stops an execution. (One, who is familiar with assert and assume, can think of (if p then e else BAD) as (assert p; e) and (if p then e else UNR) as (assume p; e).

SCC is modular and performed at definition site of each function. For example, (λv.v + 1) › tinc expands to:

λx1. let y = (λv.v + 1) (let x = x1 in if x > 0 then x else UNR') in
    if y > 0 then y else BAD(2,5,“inc”)

At the definition site of a function, f = e, we assume f’s precondition holds and assert its postcondition. If all Bads in e ▷ t are not reachable, we know f satisfies its contract t. One way to check reachability of BAD is to symbolically simplify the fragment. In the above case, inlining x, we get:

λx1. let y = (λv.v + 1) (if x1 > 0 then x1 else UNR') in
    if y > 0 then y else BAD(2,5,“inc”)

Unlike [39] in a lazy setting, we cannot apply beta-reduction in a strict language if an argument is not a value as it may not preserve the semantics. In this paper, besides symbolic simplification, we collect contextual information in logical formula form and consult an SMT solver to check the reachability of BAD. An SMT solver usually deals with formulae in first order logic (FOL), §5 gives the details of the generation of formulae in FOL. As an overview, we present formulae in higher order logic (HOL). For the two subexpressions of the RHS of y, we have:

λv.v + 1

if x1 > 0 then x1 else UNR

One can think of the existentially quantified x2 (and x3) denoting the expression itself. For the RHS of y, we have logical formula:

∀y, (∃x2, (∀v, x2(v) = v + 1) ∧ (∃x3, (x1 > 0 ⇒ x3 = x1) ∧ (not(x1 > 0) ⇒ false)) ∧ y = x2(x3))

We check the validity of ∀x1, Q1 ⇒ y > 0 by consulting an SMT solver. As ∀x1, Q1 ⇒ y > 0 is valid, we know the BAD(2,5,“inc”) is not reachable, thus inc satisfies its contract.

Consider the function f1 and its contract t1 in §1. So f1 ▷ t1 is (λg.(g 1) – 1) › ([x | x ≥ 0] → [y | y ≥ 0]) → [z | z ≥ 0], which expands to:

λx1. let z = (λg.(g 1) – 1)
    (λx2. let y = x1 (let x = x2 in
        if x ≥ 0 then x
        else BAD(4,5,f1))
    in if y ≥ 0 then y else UNR') in
    if z ≥ 0 then z else BAD(4,5,f1)

After applying some conventional simplification rules, we have:

R1: λx1. let z = let y = x1 in
    if y ≥ 0 then y - 1 else UNR'
    if z ≥ 0 then z else BAD(4,5,f1)

We see that the inner BAD(4,5,f1) has been simplified away, because x = x2 = 1 and if 1 ≥ 0 then 1 else BAD(4,5,f1) is simplified to 1. As we cannot prove ∀x1, ∀z2, (∃y, y = x1 1 ∧ (y > 0 ⇒ z = y - 1)) ⇒ z ≥ 0 to be valid, the other BAD(4,5,f1) remains. We can either report this potential contract violation at compile-time or leave this residual code R1 for DCC to achieve hybrid checking.

Hybrid contract checking (HCC) performs SCC first and runs the residual code as in DCC. In SCC, f1 ▷ t1 checks whether f1 satisfies its postcondition by assuming its preconditions hold. At each call site of f1, we wrap the function with §. For example:

contract f3 = { v | v >= 0 }
let f3 = f1 zut
where \( zut \) is a difficult function for an SMT solver and \( zut \)'s contract is \( \{ x : \text{true} \} \). Say \( zut \subseteq \{ x : \text{true} \} = zut \), then we have the term \( \text{f3} \triangleright \text{f3} \) to be:

\[
(f1 \circ \text{f1}) \triangleright \text{f1}) \triangleright \{ v | v > 0 \}
\]

which requires \( f3 \) to satisfy \( f1 \)'s precondition and assumes \( f1 \) satisfies its postcondition because \( f1 \triangleright \text{f1} \) has been checked. During SCC, a top-level function is never inlined. We do not have to know its detailed implementation at its call site as it has been guarded by its contract with \( f \triangleright t \). The \( \text{f3} \triangleright \text{f3} \) expands to:

\[
\begin{align*}
\text{let } v &= \text{let } z = \text{f1} \\
(\lambda x_2. \text{let } y = \text{zut} (\text{let } x = x_2 \text{ in } \text{if } x \geq 0 \text{ then } x \text{ else UNR}(7,10,\text{"f1"})) \text{ in } \\
&\text{if } y \geq 0 \text{ then } y \text{ else BAD}(7,10,\text{"f1"}) \text{ in } \\
&\text{if } v \geq 0 \text{ then } v \text{ else BAD}(7,10,\text{"f1"})
\end{align*}
\]

As \( c \) is dual to \( \triangleright \), the RHS of \( v \) is actually a copy of the earlier \( f1 \triangleright \text{f1} \) but swapping the BAD and UNR and substituting \( x_1 \) with \( zut \). We now know the source location of the call site of \( f1 \) and its caller’s name, the UNR\(^7\), becomes BAD\((7,10,\text{"f1"})\) and the BAD\((4,5,\text{"f1"})\) becomes UNR\((7,10,\text{"f1"})\). At definition site where the caller is unknown, we use the location of \( f1 \), i.e. \((4, 5, zut)\). Once its caller is known, we use \((7, 10, zut)\). It is easy to get source location, which is for the sake of error message reporting. So we do not elaborate the source location further.

As an SMT solver says valid for \( \forall v: (\exists z \geq 0 \land v = z) \Rightarrow v \geq 0 \), the \( \text{f3} \triangleright \text{f3} \) can be simplified to (say R2):

\[
\begin{align*}
\text{let } z &= \text{f1} \\
(\lambda x_2. \text{let } y = \text{zut} (\text{let } x = x_2 \text{ in } \text{if } x \geq 0 \text{ then } x \text{ else UNR}(7,10,\text{"f1"})) \text{ in } \\
&\text{if } y \geq 0 \text{ then } y \text{ else BAD}(7,10,\text{"f1"}) \text{ in } \\
&\text{if } v \geq 0 \text{ then } v \text{ else BAD}(7,10,\text{"f1"})
\end{align*}
\]

One BAD remains. We can either report this potential contract violation at compile-time or continue a DCC. For SCC, we have checked \( \triangleright \text{f1} \), but for DCC, to invoke \( \triangleright \text{f1} \), we must use the residual code R1. However, the UNR clauses are useful for SCC, but redundant for DCC. We can remove UNRs with a simplification rule:

\[
(\text{if } e_0 \text{ then } e_1 \text{ else UNR}) \Rightarrow e_1 \quad \text{[rmUNR]}
\]

(We shall explain why it is valid to apply this rule even if \( e_0 \) may diverge or crash in [6]. Intuitively, \( UNR \) is indeed unreachable and \( e_0 \) has been checked before this program point.) Applying the rule [rmUNR] to R1 and R2 and simplify a bit, we get:

\[
\begin{align*}
f1_f &= \lambda x_1. \text{let } z = (\text{let } y = (x_1 1) \text{ in } y - 1) \text{ in } \\
f2_f &= \lambda x_2. \text{let } y = \text{zut} x_2 \text{ in } \text{if } y \geq 0 \text{ then } y \text{ else BAD}(7,10,\text{"f1"})
\end{align*}
\]

respectively, which is the residual code being run. We show in [6] that HCC blames a function \( f \), iff DCC blames \( f \).

**Summary**

Given a definition \( d = e \) and a contract \( t \), to check \( e \) satisfies \( t \) (written \( e \triangleright t \)), we perform these steps. (1) Construct \( e \triangleright t \). (2) Simplify \( e \triangleright t \) as much as possible to \( e' \), consulting an SMT solver when necessary. (3) If no BAD is in \( e' \), then there is no contract violation; if there is a BAD in \( e' \) but no function call in \( e' \), then it is definitely a bug and report it at compile-time; if there is a BAD and function call(s) in \( e' \), then it is a potential bug. (4) For each function \( f \), create its residual code \( f_e \) by simplifying \( e' \) with the rule [rmUNR], and run the program with each \( f \) being replaced by \( f' \).

### 3. The language

The language presented in this paper, named M, is pure and strict, a subset of OCaml, including parametric polymorphism.

#### 3.1 Syntax

```
\[
x, f \in \text{Variables} \quad T \in \text{Type constructors} \\
K \in \text{Data constructors}
\]

\[
\begin{align*}
\text{pgm} &::= def_1, \ldots, def_n \\
\tau &::= \text{int} | \text{bool} | T \Rightarrow \tau \Rightarrow \tau \\
t &::= \{ x | p \} \\
def &::= \text{type } \alpha T = K \text{ of } \alpha T \\
r &::= \text{BAD} | \text{URR} \\
l &::= (n_1, n_2, \text{String}) \\
val &::= n | x | r | K \text{ val } | \lambda x^\ast. e \\
tv &::= n | x | K \text{ val } \\
tveal &::= tv | \lambda (x^\ast). e
\end{align*}
\]

Figure 1: Syntax of the language M

Figure 1 gives the syntax of language M. A program contains a set of data type declarations, contract declarations and function definitions. Expressions include variables, lambda abstractions, applications, constructors and match-expressions. Basic types such as int and bool are data types with no parameter. We have top-level let rec, but for the ease of presentation, we omit local let rec. (It is possible to allow local let rec by either assuming that a local recursive function is given a contract or using contract inference [22] to infer its contract. Even if [22] is not modular, it is good enough to infer a contract for a local function.) Pairs are a special case of constructed terms, i.e. \( (e_1, e_2) \) is Pair of \( ('a, 'b) \). A local let-expression \( \text{let } x = e_1 \text{ in } e_2 \) is a syntactic sugar for \( \lambda (x: e_1). e_2 \). An if-expression \( \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \) is syntactic sugar for match \( e_0 \) with \{ true -> e_1, false -> e_2 \}.

We assume all top-level functions are given a contract. Contract checking is done after the type checking phase in a compiler so we assume all expressions, contexts and contracts are well-typed and use its type information (presented as a superscript, e.g. \( e^\tau \) or \( t^\tau \)) whenever necessary. Type checking material is omitted, but can be found in [40].
The two contract exceptions (also called blames) BAD and UNR are adapted from [41]. They are for internal usage, not visible to programmers. The label l contains information such as function name and source location, which is useful for error reporting as well as for examination of the correctness of blaming. But we may omit the label l when it is not the focus of the discussion.

It is possible for programmers to write:

```
let head xs = match xs with
    | [] -> raise Emptylist
    | x:1 -> x
```

where `raise : ∀α. Exception → α`. The Exception is a built-in data type for exceptions and `Emptylist` type Exception.

As we do not have `try-with` in language M (leaving it as future work), a preprocessing converts `raise Emptylist` to BAD `head`.

We have four forms of contracts. The p in a predicate contract `(x | p)` refers to a boolean expression in the same language M. Dependent function contracts allow us to describe dependency between input and output of a function. For example, let `head xs = match xs with
    | [] -> raise Emptylist
    | x:1 -> x
` when it is not the focus of the discussion.

\[\text{Definition 2 (Crash)}\]

A closed term e crashes if e →* BAD.

Our framework only guarantees partial correctness. A diverging program does not crash.

\[\text{Definition 3 (Diverges)}\]

A closed expression e diverges, written e↑, if either e →* UNR, or there is no value val such that e →* val.

At compile-time, one decidable way to check the safety of a program is to see whether the program is syntactically safe.

\[\text{Definition 4 (Syntactic safety)}\]

A (possibly-open) expression e is syntactically safe iff BAD β. Similarly, a context C is syntactically safe iff BAD C β. C.

The notation BAD β e means BAD does not syntactically appear anywhere in e, similarly for BAD β C. For example, `λx.x` is syntactically safe while `λx.(BAD,x)` is not.

\[\text{Definition 5 (Crash-free Expression)}\]

A (possibly-open) expression e is crash-free if:

\[∀C. \text{BAD} \not{β} e \text{ and } (C[e])^{\text{bool}} \implies C[e] \not{α} \text{ BAD}\]

The notation `(C[e])^{\text{bool}}` means C[e] is closed and well-typed. The quantified context C serves the usual role of a probe that tries to provoke e into crashing. Note that a crash-free expression may not be syntactically safe, e.g. `Ax.1f x > x ≥ 0 then x + 1 else BAD`.

\[\text{Lemma 1 (Syntactically safe expression is crash-free)}\]

\[e \text{ is syntactically safe } \implies e \text{ is crash-free}\]

For ease of presentation, when we do not give label l to BAD or UNR, we mean BAD or UNR for any l. Moreover, expressions BAD β and UNR β are closed expressions even if l is not explicitly bound.

\section{Contracts}

Inspired by [41], we design contract satisfaction and checking algorithm for a strict language. As diverging contracts make dynamic contract checking unsound (explained in §4.3) and we do hybrid checking, we focus on total contracts.

\[\text{Definition 6 (Total contract)}\]

A contract t is total iff

\[t \equiv (x | p) \text{ and } λx.p \text{ is total (i.e. crash-free, terminating)} \text{ or } t \equiv (x : t_1 → t_2) \text{ and } t_2 \text{ is total and for all } val_1 ∈ t_1, t_2[val_1/x] \text{ is total} \text{ or } t \equiv (x : t_1, t_2) \text{ and } t_1 \text{ is total and for all } val_1 ∈ t_1, t_2[val_1/x] \text{ is total} \text{ or } t \equiv \text{any}\]

Our definition of total contract is different from that in [7], but close to the crash-free contract in [41] with an additional condition that `λx.p` is a terminating function. For example, contract `(x | x ≠ [] ) → (y | head x > y)` is total in our framework because `head x` does not crash for all x satisfying `(x | x ≠ [])`. Such a contract is not total in [7] because a contracting function head is called in a predicate contract.

\[\text{4.1 A semantics for contract satisfaction}\]

We give the semantics of contracts by defining “e satisfies t” (written e ∈ t) in Figure 3 inspired by [7, 41]. Here are some consequences: (1) a divergent expression satisfies any contract, hence all contracts are inhabited; (2) only crash-free expression satisfies a predicate contract; (3) any expression satisfies contract any; (4) BAD only satisfies contract any.

One difference from [41] is that, we do not allow `p[e/x]` in [A1] to diverge while [41] allows because they only do static checking. We support dependent tuple contracts, that are not in [7, 41]. One difference from [7] is that, they say that a crashing expression does not satisfy any contract; we say that a crashing expression satisfy the universal contract any. Having a top ordering contract is debated in [12] where a subcontract ordering is defined below.
For a well-typed expression $e$, define $e \in t$ thus:

$$e \in \{x \mid p\} \iff e \uparrow \text{ or (}e\text{ is crash-free and }\forall x \in x.\ e[x] \rightarrow \top\text{ true})$$

$$e \in x: t_1 \rightarrow t_2 \iff e \uparrow \text{ or } (e \rightarrow^* \lambda x. e_2 \text{ and }\forall val_1 \in t_1. (e val_1) \rightarrow t_2[\text{val}_1/x])$$

$$e \in (x: t_1, t_2) \iff e \uparrow \text{ or } (e \rightarrow^* (\text{val}_1, \text{val}_2) \text{ and }\forall val_1 \in t_1 \text{ and }\forall val_2 \in t_2[\text{val}_1/x])$$

$$e \in \text{Any} \iff \text{true}$$

*Figure 3: Contract Satisfaction*

**Definition 7** (Subcontract). For all closed contracts $t_1$ and $t_2$, $t_1$ is a subcontract of $t_2$, written $t_1 \subseteq t_2$, if $\forall e. e \in t_1 \Rightarrow e \in t_2$

It is obvious that $\text{Any}$ is useful in a lazy language [41] as we may want to ignore some subcomponents of a constructor. It is also useful to have contract $\text{Any}$ for a strict language. Consider:

contract fail = Any
let fail = raise Error

where $\text{Error}$ has type Exception. One can think of $\text{Any}$ as $\forall x.\ x$. In [7] and other refinement type checking framework [5, 25, 37], they give function like fail a contract function $\{x \mid \text{false}\} \rightarrow \{x \mid \text{true}\}$ so that the precondition $\{x \mid \text{false}\}$ allows their system to blame all the callers of fail. This is somewhat ad hoc. More discussion on the contract $\text{Any}$ can be found in [40].

### 4.2 The wrappers

As mentioned in §2, the essence of contract checking is the two wrappers $\triangleright$ and $\triangleleft$ which are dual to each other (defined in Figure 4). We omit the labels for $\triangleright$ and $\triangleleft$ whose full versions are $\triangleright_i$ and $\triangleleft_i$ respectively. The wrapped expression $e \triangleright_i t$ expands to a particular expression, which behaves the same as $e$ except that it raises blame $r_1$ if $e$ does not obey $t$ and raise $r_2$ if the wrapped term is used in a way disobeying $t$.

$$e \triangleright t = e \triangleright_i \frac{r_1}{r_2} t$$

From [P1] to [P3], if $e$ crashes, the wrapped term crashes; if $e$ diverges, the wrapped term diverges. Whenever an $r_i$ is reached, we know the property $p$ does not evaluate to true (as in [P1]). Contents in Figure 3 and 4 are defined such that Theorem 1 holds.

**Theorem 1** (Sound-and-completeness of contract checking). For all closed expression $e'$, closed and total contract $t'$,

$$(e \triangleright t) \text{ is crash-free } \iff e \in t$$

The superscript $\triangleright$ says both $e$ and $t$ are well-typed and have the same type $\tau$. The full proof of Theorem 1 is in [40]. Basically, we do the kind of proofs in [42] but for a strict language. In practice, we only need Theorem 2, i.e. one direction of Theorem 1.

**Theorem 2** (Soundness of contract checking). For all closed expression $e'$, closed and terminating contract $t'$,

$$(e \triangleright t) \text{ is crash-free } \Rightarrow e \in t$$

Note that if $t$ is terminating and $e \triangleright t$ is crash-free, then $t$ is total. Unlike [13], which assumes there is no exception from a contract itself, our contract checking algorithm helps programmers to ensure it by detecting exceptions in contracts themselves. The term $t_i([v \triangleright_i t_1/x])$ in [P2] and [P3] says that, we wrap each subterm (function) call in a contract with its contract so that, if there is any contract violation in a contract, we report this error. For example:

contract $f = k: \{x \mid x > 0 \} \rightarrow \{y \mid y > 0 \}$

let $g = g 2$
let $t 2 = f (\text{fun } x \rightarrow x)$

a contract violation occurs in $\{x \mid k 0 > -1\}$ because the call $k 0$ fails k’s precondition $\{x \mid x > 0\}$. As addressed in [10], we should blame the contract. We omit passing around the name of the contract in this paper as our focus is to check the reachability of BAD instead of $r_i$ in order to make their proof go through. Our proof [40] is different from that in [7].

Given $f = e$, where $e$ is open, and a (possibly open) contract $t$, to check $\Gamma \vdash e \in t$ where $\Gamma$ is an environment mapping a variable to its contract, we check $e((f_i \triangleright t_i)/f_i) \triangleright t((f_i \triangleright t_i)/f_i)$ where $f_i$ are free variables in $e$ (or $t$) and are in the domain of $\Gamma$. Note that $f_i$ can also be a recursive call $f$ and $f \triangleright e t$ in $e((f_i \triangleright t_i)/f_i) \triangleright t((f_i \triangleright t_i)/f_i)$ is like an induction hypothesis. As $e((f_i \triangleright t_i)/f_i) \triangleright t((f_i \triangleright t_i)/f_i)$ is closed, we only have to reason close expressions and contracts, similar to [41].

### 4.3 Terminating contracts

We want $p \in \{x \mid p\}$ to be terminating because a divergent contract hides crashes. For example:

let rec $\text{loop } x = \text{loop } x$
contract $fb = (x \mid \text{loop } x) \rightarrow \{y \mid \text{true}\}$
let $fb x = \text{head } []$

$fb \triangleright t 2$ is $\lambda x.((\text{loop } x) \rightarrow \{y \mid \text{true}\})$ (if $\text{loop } x$ then $x 1$ else BAD)), which diverges whenever applied because of the loop. However, the function $fb$ is not crash-free

We only have to prove termination of functions used in contracts, not all the functions in a program. We can adapt ideas in [4, 28, 36] to build an efficient automatic termination checker.

### 5. Static contract checking and residualization

Thanks to the ground-breaking higher order contract wrappers $\triangleright_i$ (first introduced in [13]), which makes the analysis of higher order program much easier. From Theorem 2, all we need is to show that $e \triangleright t$ is crash-free. That is to check the reachability of BAD as each BAD signals a contract violation. We can symbolically simplify $e \triangleright t$ as much as possible to $e'$ and check occurrence of BAD in $e'$.

We introduce an SL machine (Figure 5) which combines symbolic simplification and contextual information (ctx-info) synthesis
Figure 5: SL machine

Figure 6: Simplification Rules

with logical formulae. The novelty of our work is to combine them in a way to achieve verification, blaming and residualization in one go. The SL machine takes an expression e and produces its semantically equivalent and simplified version. A 4-tuple $\langle H \mid e \mid S \mid L \rangle$
is pronounced simplify and a 4-tuple \( \langle H \mid e \mid S \mid L \rangle \) is pronounced rebuild where
- \( H \) is an environment mapping variables to trivial values;
- \( e \) is the expression under simplification (or being rebuilt);
- \( S \) is a stack which embodies the simplification context, or continuation that will consume a simplified expression;
- \( L \) is a logical store which contains the ctx-info in logical formula form; its syntax is
\[
L ::= \emptyset \mid \forall x:\tau, L \mid \phi, L
\]
where \( \phi \) is a predicate in Figure 7.

The job of the SL machine is to simplify an expression as much as possible, consulting the logical store when necessary; when it cannot simplify the expression further, it rebuilds the expression.

**Theorem 3 (SL machine terminates).** For all expression \( e \), there exists an expression \( a \) such that \( \langle \emptyset \mid e \mid \emptyset \mid \emptyset \rangle \leadsto^* a \), where \( \equiv \) is the extensional equality.

Intuitively, SL machine behaves like CEK machine [15], but rebuilds an expression and does not inline top-level functions. As we do not have local let-rec in our language, only inline trivial values and also call SMT solver Alt-ergo with an option “stop (time-bound)” or “steps (bound)” to make sure the SMT solver terminates, there is no element causing non-termination.

**Theorem 4 (Correctness of SL machine).** For all expression \( e \), if \( \langle \emptyset \mid e \mid \emptyset \mid \emptyset \rangle \leadsto^* a \), then \( e \equiv a \).

The SL is designed in a way such that the simplified \( a \) preserves the semantics of the original expression \( e \). The proof of Theorem 4 (in [40]) uses the fact that, if there exists \( e_3 \) such that \( \langle H \mid e_1 \mid S \mid L \rangle \leadsto^* \langle H \mid e_3 \mid S \mid L \rangle \), and \( \langle H \mid e_2 \mid S \mid L \rangle \leadsto^* \langle H \mid e_3 \mid S \mid L \rangle \), then \( e_1 \equiv e_2 \) (See Definition 1 for \( \equiv \)).

**Theorem 5 (Soundness of static contract checking).** For all closed expression \( e \), closed and terminated contract \( t \)
\[
\langle \emptyset \mid e \mid t \mid \emptyset \rangle \leadsto^* e' \quad \text{and} \quad \text{BAD} \not\subseteq e' \quad \Rightarrow \quad e \in t
\]
**Proof.** By Theorem 4, Lemma 1 and Theorem 2. \( \Box \)

### 5.1 The SL machine

In Figure 5, the constant \( n \) and blame \( r \) cannot be simplified further, thus being rebuilt as shown in [S-const] and [S-exn] respectively. One might ask why we rebuild rather than return a blame. There are two reasons: (a) it gives more information for static error reporting, i.e. we know conditions leading to a reachable BAD; (b) do we hybrid contract checking, we want to send the residual code with undischarged blame to a dynamic checker.

As we perform symbolic simplification rather than evaluation (as in CEK machine [15]), we only put a variable in the environment \( H \) if it denotes a trivial value. A variable denoting a top-level function is not put in \( H \). Variables in \( H \) are inline by [S-var1] while variables not in \( H \) are rebuilt by [S-var2].

Each element on the stack is called a stack frame where the hole \( \bullet \) in a stack frame refers to the expression under simplification or being rebuilt. We use \( a \) to represent an expression that has been simplified. The syntax of a stack frame \( S \) is
\[
s ::= [\_ | (e \_) :: s | (e \bullet) :: s | (\lambda x. \bullet) :: s | \text{let } x = e \in e | (\text{match } \bullet \text{ with } \_ \rangle :: s | \text{let } x = e \in \bullet | (\text{match } e_0 \text{ with } \_ \rightarrow (e, S, L) :: s)
\]

The transitions [S-app], [S-match] and [S-K] implement the context reduction in Figure 2. The transitions [S-letL], [S-matchL], [S-letR], [S-matchR], [S-match-match], [S-match-let] implement the conventional simplification rules in Figure 6. Here, \( \_ \rightarrow \_ \) abbreviates a sequence of \( x_1, \ldots, x_n \). We use let instead of lambda for easy reading. Rules [letL] and [matchL] push the argument into the let-body and match-body respectively. Rules [letR] and [matchR] push the function into the let-body and match-body. The rules [match-match] and [match-let] are to make an expression less nested. Rule [K-match] allows us to simplify
\[
\text{match } Some \ e \ with \ \{ \text{Some } x \rightarrow 5; \text{None } \rightarrow \text{BAD} \}
\]
where \( e \) is a crash-free expression, not a value) to let \( x = e \in 5 \) which is crash-free.

What does rebuild do? If the stack is empty (R-done), which indicates the end of the whole simplification process, we return the expression. Otherwise, we examine the stackframe. By [E-exn], the transition \( \text{[R-t]} \) rebuilds UNR (or BAD) with the rest of the stack. After we finish simplifying one subexpression, we start to simplify another subexpression (e.g. [R-fun]). When all subexpressions are simplified, we rebuild the expression (e.g. [R-lam] and [R-app]). If current simplified expression is a trivial value and we have stack frame lambda on \( S \), we use [R-beta]; together with [S-var1], they implement a beta-reduction [E-beta]. Bound variables are renamed when necessary.

The logical store \( L \) captures all the ctx-info up to the program point being simplified. (We use if-expression to save space, but refer to match-transitions.) Consider:
\[
\langle H \mid \text{if } x > 0 \text{ then } (if x + 1 > 0 \text{ then } 5 \text{ else BAD}) \rangle \quad \text{and} \quad \text{BAD} \not\subseteq e' \quad \Rightarrow \quad e \in t
\]

Before applying the transition [R-s-save], we check whether \( x > 0 \) or not(\( x > 0 \)) is implied by \( L \) to see whether the transition [R-s-match] can be applied. The transition [R-s-match] implements [E-match], where the side condition “if \( 3(K \not\rightarrow \tau) \), \( \tau \not\rightarrow \emptyset[\lambda x.\tau] \)” checks if there is any branch \( K \not\rightarrow \) that matches the scrutinee \( a \). But the current information in \( L \) is not enough to show the validity of either \( x > 0 \) or not(\( x > 0 \)). By [R-s-save], we convert this scrutinee to logical formula with \( \emptyset[\lambda x.\tau] \) (explained later) and put it in \( L \) and simplify both branches. Note that we put \( x > 0 \) in \( L \) for the true branch while not(\( x > 0 \)) for the false branch.

\[
(H \mid \text{if } x + 1 > 0 \text{ then } (\text{if } x > 0 \text{ then } e) \text{ else BAD} \rangle \quad \text{and} \quad \text{BAD} \not\subseteq e' \quad \Rightarrow \quad e \in t
\]

In the true branch, after a few steps, we rebuild the scrutinee \( x + 1 > 0 \). In this case, \( \forall x : \text{int}, x > 0 \Rightarrow x + 1 > 0 \) is valid. By [R-match], we take the true branch, which is a constant 5. As both 5 and UNR cannot be simplified further, we rebuild them by [S-const] and [S-unr] respectively and obtain:
\[
\langle H \mid 5 \rangle \quad \text{and} \quad \text{BAD} \not\subseteq e' \quad \Rightarrow \quad e \in t
\]

By [R-match], we combine both simplified branches to rebuild the match-expression:
\[
\langle H \mid \text{if } x > 0 \text{ then } 5 \text{ else UNR} \rangle \quad \text{and} \quad \text{BAD} \not\subseteq e' \quad \Rightarrow \quad e \in t
\]
We continue to rebuild the expression by [R-lam]:
\[
\langle \mathcal{H}, \lambda x. \text{if } x > 0 \text{ then } 5 \text{ else UNR} \rangle \cdot \langle \mathcal{H}, \exists x. \text{int} \rangle
\]
and terminate (by [R-done]) with a syntactically safe expression:
\[
\lambda x. \text{if } x > 0 \text{ then } 5 \text{ else UNR}
\]
Besides [R-s-save], another transition that saves ctx-info to \(\mathcal{L}\) is [R-let-save]. Consider an example:
\[
\lambda v. \text{let } y = v + 1 \text{ in if } y > v \text{ then } y \text{ else BAD}
\]
After a few simplification steps, we have:
\[
\langle \mathcal{H}, v + 1 \rangle \cdot \langle \text{let } y = \bullet \text{ if } y > v \text{ then } \lambda (\mathcal{H}, v) \rangle
\]
The rule [R-let-save] saves the information \(y = v + 1\) to \(\mathcal{L}\), which allows us to check the validity of the scrutinee \(y > v\) later.
\[
\langle \mathcal{H}, v + 1 \rangle \cdot \langle \text{let } y = v + 1 \text{ in } \lambda (\mathcal{H}, v) \rangle \cdot \langle \mathcal{H}, v \rangle
\]
Since \(\forall v : \text{int}, \forall y : \text{int}, y = v + 1 \Rightarrow y > v\) is valid, by [R-s-match], we only need to simplify the truth branch:
\[
\langle \mathcal{H}, v \rangle \cdot \langle \text{let } y = v + 1 \text{ in } \lambda (\mathcal{H}, v) \rangle \cdot \langle \mathcal{H}, v \rangle
\]
which leads to the final result \(\lambda v. \text{let } y = v + 1 \text{ in } y\), which is syntactically safe.

## 5.2 Logicization

We now explain the mysterious conversion \([\lambda x. f]_\mathcal{L}\), which we call logicization. Figure 7 gives the abstract syntax of the logical formula supported by an SMT solver named Alt-ergo, which is an automatic theorem prover for polymorphic first-order logic modulo theories. It uses classical logic and assumes all types are inhabited. First, Alt-ergo allows data type declaration e.g.

\[
\text{type } ('a, 'b) \text{ arrow } = \{ (x, y) \mid x : 'a \rightarrow 'b \}
\]

The function \(\text{len}\) computes the length of a list and the function \(\text{append}\) appends two lists. Let \(\text{eq}\) and \(\text{ord}\) stand for the definition and contract of \(\text{append}\) respectively. Applying only simplification rules (including reduction rules) to \(\text{eq}\) and \(\text{ord}\), we get (R3):
\[
\lambda v, \text{ord}. \text{match } v, \text{ord} \text{ with } \{ [ ] \rightarrow y \}
\]

\[
\text{let } x : u \rightarrow x : \text{append } u y
\]

The simplification approach in [39] and the model checking approach in [34] involve inlining top-level functions, while we do not. Instead, we axiomatize top-level function definitions called in contracts and lift expressions under checking to logic level and consult an SMT solver. The challenge is to deal with non-total expressions (e.g. BAD) in our source code. In the literature of converting functional code (in an interactive theorem prover) to SMT formula \([1, 6, 9, 29]\), they convert expression to a logical form directly. In [1], given a non-recursive function definition \(f = \epsilon\), they first \(\lambda z. \text{expand } c \rightarrow f = \lambda x_1 \ldots x_n. c' \) where \(c'\) does not contain \(\lambda\); if it is a recursive function, they assume \(\epsilon\) is in a particular form such that all lambdas are at top-level and the function performing an immediate case-analysis over one of its arguments. Then, they...
form \( \forall \varphi, f(x_1, \ldots, x_n) = [e'] \) where \([\cdot]\) converts an expression to logical form. (On the other hand, [6] uses \( \lambda \)-lifting method: \( \lambda \)-abstractions are translated from inside out, each \( \lambda \)-abstraction is replaced by a call to a newly defined functions. That is to form \( \forall \varphi, f_n(x_1, \ldots, x_n) = [e'][x_1; \ldots; x_n] = f_1(x_1) \).) This is fine for converting total terms, e.g. \([5] = 5) and \([x] = x. \)

Our task is to examine the scrutinee of a goal app_1 : forall v1,v2:'a list. v1 = nil -> apply(len,v2) = apply(len,v1) + apply(len,v2)

\( \exists x. \forall \varphi, \exists v. \forall \varphi, \exists v. \) For example, the axiom \( \exists x. : \forall \varphi, \exists v. : \forall \varphi, \exists v. \) is generated by:

\[ \forall \varphi, \exists v. : \forall \varphi, \exists v. \]

To make an SMT solver’s life easier (i.e. multiple small axioms are non-total terms) nor \( \lambda \)-lifting, and yet we allow arbitrary forms of recursive functions. We have such flexibility because we convert \( \lambda \)-abstraction and partial application directly with the help of apply.

(Also note that our logicization \([\cdot]\) can also produce HOL formula for interactive proving by replacing \((\text{apply}(f, x)) \) by \((f(x))\) and not converting the types.) No logicization work in the literature (including \([6, 9, 29, 35]\)) deal with non-total terms. The work \([6]\) uses approaches in \([9, 29]\) to deal with polymorphism while Alt-ergo itself supports polymorphism.

Our framework can systematically generate Alt-ergo code, like below, to show that those B\&D in R3 are unreachable.

\[ \begin{align*}
\text{logic len: ('a list, int) arrow} \\
\text{logic append: ('a list, ('a list, 'a list) arrow) arrow} \\
\text{axiom len_def_1: forall s: 'a list. s = nil -> apply(len,s) = 0} \\
\text{axiom len_def_2: forall s: 'a list. s = cons(x,1) -> apply(len,s) = 1 + apply(len,1)} \\
\text{goal app_1: forall v1,v2: 'a list. v1 = nil -> apply(len,v2) = apply(len,v1) + apply(len,v2)} \\
\text{goal app_2: forall v1,v2: 'a list. v1 = cons(x,1) -> apply(len,apply(append,v1,v2))}
\end{align*} \]

To make an SMT solver’s life easier (i.e. multiple small axioms are better than one big axiom), we have two axioms for \( \text{len} \), one for each branch, which are self-explanatory. As a constructor is always fully applied, we do not encode its application with apply. The \( \rightarrow \) (in axioms and goals) is a logical implication.

\[ \begin{align*}
\text{Theorem 6 (Logicization for axioms).} \quad \text{Given definition } f = e', \exists f: \tau, [e'] \text{ is valid.} \\
\text{The next, what query (i.e. goal) shall we make? All we want is to check if the branch leading to B\&D is reachable or not. So our task is to examine the scrutinee of a match-expression. For example, in the goal app_1, the ctx-info v1nil is from the pattern matching \( \text{match} \) with \{\{\} -> \ldots\}; the query is \( \text{apply(len,v2) = apply(len,v1) + apply(len,v2)} \). The goal app_1 states the ctx-info \( L \) implies the scrutine. We have \( L = \forall v1 : 'a list, \forall v2 : 'a list. v1, v2 = \text{nil by [S-lam] and [R-s-save]. The scrutinee is \{len v2 = len v1 + len v2\}true. That is, we want to check whether len v2 = len v1 + len v2 is equivalent to true. Alt-ergo says valid for both goals. Thus, we know both B\&D and B\&D' are not reachable.} \]

\[ \text{Theorem 7 (Validity preservation: logicization for goals). For all (possibly open) expression } e', \exists f: \tau, \text{ if } \forall v'e(e): \tau, [e'] \text{ is valid and } e' \rightarrow e \text{ for some } e', \text{ then } \forall v'e(e'): \tau, [e'] \text{ is valid.} \]

There are a few things to note about logicization.

\[ \text{Syntax abbreviation} \quad \text{The Alt-ergo syntax} \]

\[ \text{logic } x: \text{ty}; \text{ axiom } a_i: \phi_i; \text{ goal } g_j: \phi_j \]

is semantically the same as \( \forall x: \text{ty}, \text{axiom } a_i: \phi_i, \text{ goal } g_j: \phi_j \) where \( \phi \) means a conjunction of a set of logical formulae.

\[ \text{Only functions called in contracts are converted to Alt-ergo axioms} \quad \text{To check a function (say append) satisfies its contract, we do not convert its definition to axioms. As the wrappers \( \circ, \circ \) have inserted contract checking obligation appropriately such that function calls (including recursive calls) are guarded by their contracts.} \]

\[ \text{Crashing functions called in contracts} \quad \text{In Figure 10, there are two conversions for B\&D, true for axioms and false for goals. For example, we may have:} \]

\[ \text{contract } g = (x \mid x /= 0) \rightarrow (y \mid head x > y) \]

In this case, the contract of \( g \) is total even if a partial function head is called in the contract. The logicization of head gives:

\[ \text{logic head: ('a list, 'a) arrow} \]

\[ \text{axiom head_def_1: forall x: 'a list. x=[] -> true} \]
To illustrate the idea with less cluttered form, we omit the conversion head_def_1
The key thing is that the axiom head_def_1 is not a false axiom, it just does not give us any information, which is what we want.

Contracts that diverge Suppose divergent functions loop and nloop are used in a contract.
let rec loop x = loop x
let rec nloop x = not (nloop x)

Logicization gives:
logic loop : 'a -> 'a
axiom loop_def_1 : forall x: 'a.
  apply(loop, x) = apply(loop, x)
logic nloop : bool -> bool
axiom nloop_def_1 : forall x:bool.
  apply(nloop, x) = not (apply(nloop, x))

Axiom loop_def_1 is same as stating true, which does not hurt. But axiom nloop_def_1 is same as stating false, which we must not allow. Fortunately, we only convert functions used in contracts that can be proved terminating (in §4.3) to axioms. We will not generate the axiom nloop_def_1.

BAD and UNR For goals, the ∈j collects ctx-info before a scrutinee of a match-expression, thus, $[BAD] = [UNR] = [false]$, which implies everything. For example:

\[
\text{fun } x \rightarrow \text{let } y = \text{if } x > 0 \text{ then } x \text{ else UNR in}
\]

\[
f y + 1 > 0 \text{ then } y + 1 \text{ else BAD}
\]

The ctx-info $\mathcal{L}$ before $y + 1 > 0$ is $\forall x: \text{int, } \forall y: \text{int, } (x > 0 \Rightarrow y = x) \land (\neg (x > 0) \Rightarrow \text{false}).$ So $\mathcal{L} \Rightarrow y + 1 > 0$ is $\forall x: \text{int, } \forall y: \text{int, } (y > 0 \Rightarrow y = x) \land (\neg (x > 0) \Rightarrow \text{false}) \Rightarrow y + 1 > 0$, which is valid. It means, if $\neg (x > 0)$ holds, $y + 1 > 0$ will not be reached. Similar reasoning applies if we replace the UNR by BAD in the above example.

5.3 Discussion and preliminary experiments

One might notice that SL machine simplifies terms under lambda and the body of match-expression while we do not have such execution rules in Figure 2. As we rebuild blames and do not inline recursive functions (i.e. no crashing and no looping during simplification), SL machine does not violate call-by-value execution.

One might worry that the rule [match-match] causes exponential code explosion for static analysis (although no run-time overhead). For example, $h_1 = \text{if } a \text{ then } b \text{ else } c$, then $d$ else $c$, where $a, b, c, d, e$ are expressions. At program point $d$, the ctx-info is $(a \Rightarrow b) \land (\neg (a) \Rightarrow c)$. Applying [match-match] to $h_1$, we get: $h_2 = \text{if } a \text{ then } (\text{if } b \text{ then } d \text{ else } e) \text{ else } (\text{if } c \text{ then } d \text{ else } e)$. The $d$ is duplicated and the ctx-info for the first $d$ is $a \land b$ while for the second $d$ is $\neg(a) \land c$. With [match-match], we send smaller formula to an SMT solver (which is good for an SMT solver), but we may communicate with the SMT solver more often. From our current observation, it is quite often that the $e$ is BAD or UNR, the SL machine immediately rebuilds the blame with the rest of the stack, and we get: $c$ if $a$ then $(\text{if } b \text{ then } d \text{ else } e)$ else $c$. So $d$ is not duplicated and we have smaller formula for the SMT solver.

One advantage of the SL machine is to allow adding or removing a rule easily. In the inc example in §2, with rule [matchR], we can simplify

\[
(\lambda v. v + 1) \text{ if } x_1 > 0 \text{ then } x_1 \text{ else UNR}'\]

to if $x_1 > 0$ then $(\lambda v. v + 1) \text{ if } x_1 > 0 \text{ then } x_1 \text{ else UNR}'. \text{ As the variable } x_1 \text{ and the contract exception UNR}' \text{ are values, performing beta-reduction, we get: if } x_1 > 0 \text{ then } x_1 + 1 \text{ else UNR}'. \text{ Now, we have a logical formula (denoted by Q2):}

\[
\exists y, (x_1 > 0 \Rightarrow y = x_1 + 1) \land (\neg (x_1 > 0) \Rightarrow \text{false})
\]

which is equivalent but smaller than the Q1 in §2.

We have implemented a prototype[2] based on the source code of ocaml-3.12.1. Table 1 shows the results of preliminary experiments, which are done on a PC running Ubuntu Linux with quad-core 2.93GHz CPU and 3.2GB memory. We take some examples from [27] and OCaml stdlib and time the static checking. The column Ann gives the LOC for contract annotations.

<table>
<thead>
<tr>
<th>Program</th>
<th>Total LOC</th>
<th>Ann LOC</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro123, neg</td>
<td>23</td>
<td>4</td>
<td>0.08</td>
</tr>
<tr>
<td>McCarthy's 91</td>
<td>4</td>
<td>1</td>
<td>0.02</td>
</tr>
<tr>
<td>ack, fnhin</td>
<td>12</td>
<td>2</td>
<td>0.06</td>
</tr>
<tr>
<td>arith, sum, max</td>
<td>26</td>
<td>4</td>
<td>0.20</td>
</tr>
<tr>
<td>zipunzip</td>
<td>12</td>
<td>2</td>
<td>0.10</td>
</tr>
<tr>
<td>OCaml stdlib/list.ml</td>
<td>81</td>
<td>16</td>
<td>0.72</td>
</tr>
</tbody>
</table>

The preliminary result is promising: it checks a hundred lines of code (LOC) in a few seconds. This paper focuses on the theory of hybrid contract checking, we leave more optimization and rigorous experimentation on tuning the strength of symbolic simplification and the frequency of calling an SMT solver as future work.

6. Hybrid contract checking

We have explained with examples how SCC, DCC, HCC work in §2. Programmers may choose to have SCC only, DCC only, or HCC. In this section, we summarize their algorithm. Given a program $f_i \in t_i$, $f_i = e_i$ for $1 \leq i \leq n$. Suppose $f_i$ is the current function under contract checking; $f_j$ is a function called in $f_i$ (including $f_i$’s recursive call); $\text{SL}$ is the SL machine; $\text{rmUNR}$ implements the rule $[\text{rmUNR}]$ (mentioned earlier in §2).

\[
(f_0 \text{ then } c_1 \text{ else UNR}) \Rightarrow c_1 \text{ [rmUNR]}
\]

We have:

\[
\text{[SCC]}: \text{SL}(e_i[[f_j f_j^t]/f_j]) > t)
\]

\[
\text{[DCC]}: e_i[[f_j f_j^t]/f_j]/f_j)
\]

\[
\text{[HCC]}: f_j \Rightarrow \text{SL}(e_i[[f_j f_j^t]/f_j] f_j)/f_j)
\]

In [HCC], the residual code $f_j$’s parameter “$t$” waits for a caller’s name. For example, if an STM solver cannot prove the goal app.2 in §5.2 (although it can), recalling R3 in §5.2, the residual code append is:

\[
\lambda t. \lambda v_1, \lambda v_2. \text{match } v_1 \text{ with}
\]

\[
[] \rightarrow v_2;
\]

\[
x : = l \text{ if } \text{len } x : \text{ append } t v_2 = \text{len } v_1 + \text{len } v_2
\]

\[
\text{then } x : = \text{append } t v_2 \text{ else BAD'}
\]

which says that we only have to check postcondition for the second branch. (If all BADS are simplified away during SCC, a residual code of a function is its original definition.)

Lemma 2 (Telescoping property [7, 41]). For all expression $e$, total contract $t$, blames $r_1, r_2, r_3, r_4$, $(e^\tau \langle t \rangle)\ r_3^\tau t = e^\tau\ r_4^\tau t$.\footnote{http://gallium.inria.fr/~naxu/research/hcc.html}
Precondition of a function is checked at caller sites. An $f$ is the simplified $f \uparrow^i_j t_{fj}$, inspecting [HCC], each $f$ at caller sites is replaced by $(f \uparrow^i_j t_{fj}) \downarrow^j_i t_{fj}$, which is $(f \uparrow^i_j t_{fj}) \downarrow^j_i t_{fj}$. By the telescoping property, we have:

$$(f \uparrow^i_j t_{fj}) \downarrow^j_i t_{fj} = (f \uparrow^i_j t_{fj}) \downarrow^j_i t_{fj} \quad [T1]$$

which is the same as in DCC. This shows that [HCC] blames $f$ if and only if [DCC] blames $f$.

Moreover, [T1] justifies the correctness of applying the rule [ranUNR] because all UNRs are indeed unreachable as BAD$^i$ is invoked before UNR$^i$ for the same $t$. That is, if $p \text{ then } e_1 \text{ else } BAD^i$ is invoked before $p \text{ then } e_1 \text{ else } UNR^i$ for the same $p$, maybe different $e$. So it is safe to apply the rule [ranUNR] even if $p$ diverges or crashes because the same $p$ in $(p \text{ then } e_1 \text{ else } BAD^i)$ diverges or crashes first. It is easy to see if $t = \{x \mid p\}$. If $t = t_1 \rightarrow t_2$, then $(e \downarrow^0_j t_1 \rightarrow t_2) \downarrow^j_0 t_1 \rightarrow t_2$ expands to let $y = e$ in

$$\lambda y.0.((\lambda y_1. (y (v_1 \downarrow^0_j t_1))) (y (v_2 \downarrow^0_j t_1))) (y (v_2 \downarrow^0_j t_1)) \downarrow^j_0 t_2$$

Focusing on the BADs and UNRs above $\downarrow^i_j$, inspecting [P1] in Figure 4, BAD$^i$ is invoked before UNR$^i$ and BAD$^i$ is invoked before UNR$^i$. This holds inductively on the size of $t$ [40].

7. Related work

Contract semantics were first formalized in [7, 12] for a strict language and later in [41] for a lazy language. This paper adapt and re-formalize some of their ideas on contract satisfaction and contract checking. Detailed design deference is explained in §4.

Pre/post-condition specification using logical formulae [2, 16, 18, 35] allows programmers to existentially quantify over infinite domains or express meta-properties that are not expressible in contracts. However, such property cannot be converted to program code for dynamic checking. As automatic static checking already has its utilization, being able to convert some difficult checks to dynamic checks is practical. Refinement types and contracts can be enhanced in many ways like we did for types, e.g. sub-contract relation [12, 42], recursive contracts [7], polymorphic contracts [3]. Contracts also enjoy interesting mathematical properties [7, 12, 40, 41]. We like the idea of ghost refinement in [37] that separates properties that can be converted to program code from the meta-properties logical formulae.

One might recall the hybrid refinement type checking (HTC) [14, 25]. In theory, [17] shows that (picky/indy, i.e. our) contract checking is able to give more blame than refinement type checking in the presence of higher order dependent function contracts. That is partly why [37] invents a Kind checker to report ill-formed refinement types. As discussed in §4.2, we check $c \uparrow^i_j t$ to be crash-free in one-go and do not have to check $t$ to be crash-free separately. In practice, the $H$ and $L$ in the SL machine serve the similar purpose as the typing environment in HTC. But the symbolic simplification gives more flexibility such as teasing out the path sensitivity analysis with the rule [match-match], etc. We hope this work opens a venue to compare HCC and HTC in practice, such as the kind of properties we can verify, the speed of static checking, the size and speed of the residual code generated, etc. Notably, VeriFast [21] (for verifying C and Java code) suggests that symbolic execution is faster than verification condition generation method [2, 16].

The work [24] mixes type checking and symbolic execution. However, [24] requires programmers to place block annotations $\{ t \}$ for type checking and $\{ \}$ for symbolic execution while our SL machine systematically simplifies subterms and consults the logical store for checking at the appropriate program point. The [24] does not generate residual code while we do. Moreover, their symbolic expression is in linear arithmetics, which is more restrictive than ours.

Our approach is different from [37], which extracts proofs of refinement types from an SMT solver and injects them as terms in the generated bytecode RDCIL (like proof carrying code) during refinement type checking. It is for security purpose.

Some work [26, 27, 33, 34] suggest to convert program to higher order recursive scheme (HORS), which generates (possibly infinite) trees, and specify properties in a form of trivial automaton and do model checking to know whether HORS satisfies its desired property. Our approaches are completely different although we both do reachability checking. They work on automaton while we work on program directly. Our approach is modular (no top-level function is inlined) while theirs is not. They deal with local let rec (i.e. invariant inference) while we do not, but we could infer local contract with method in [22] or inline the local let rec function for a fixed number of times. They deal with protocol checking while we do not unless a protocol checking problem can be converted to checking the reachability of BAD. SL machine (in §5) can be used for any problem that checks the reachability of BAD in general.

The contextual information synthesis and conversion of expression to logical formula is inspired by the use of the application * in [19, 20], which makes conversion of higher order functions easier. But we use the technique in different contexts. Many papers on program verification [2, 11, 16, 31, 32, 38] focus on memory leak, array bound checks, etc. and few handle higher order functions and recursive predicates. Our work focus on more advanced properties and blame precisely functions at fault. Contract checking in the imperative world is lead by [11], which statically checks contract satisfaction at bytecode CIL level and run dynamic checking separately. Residualization has not been done in [11]. We may adapt some ideas in [21] to extend our framework for program with side effects.

8. Conclusion

We have formalized a contract framework for a pure strict higher order subset of OCaml. We propose a natural integration of static contract checking and dynamic contract checking. With SL machine, our approach gives precise blame at both compile-time and run-time in the presence of higher order functions. In near future, besides rigorous experimentation and case-studies, we plan to add user-defined exceptions; allow side-effects in program and hidden side-effects in contracts; do contract or invariant inference as [11, 22, 31] are inspiring.

Acknowledgments

I would like to thank Xavier Leroy, Francois Pottier, Nicolas Poulard, Martin Berger, Simon Peyton Jones, Michael Greenberg for their feedback.

References
