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:

```
(* val f1 : int -> int -> int *)
contract f1 = ({x | x >= 0} -> {y | y >= 0})
             -> {z | 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 postcondition: the static checker relies on the invalidity of $\forall g : \texttt{int} \rightarrow \texttt{int},\ (g\ 1) \geq 0 \Rightarrow (g\ 1) - 1 \geq 0$ while the dynamic checker evaluates `(((fun x -> x - 1) 1) - 1)` to `-1`, which violates the contract `{z | z >= 0}`. However, a dynamic checker cannot tell that the argument `(fun x -> 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 `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 $\triangleright$ that takes an expression e and its contract t and produces a term $e \triangleright t$ such that contract checks are inserted at appropriate places in e. If a contract check is violated, a special constructor $\texttt{BAD}^l$ signals the violation. As the term $e \triangleright t$ is a term in the same language as e, all we have to do is to check the reachability of $\texttt{BAD}^l$. If a `BAD` is reachable, we know a contract is violated and the label l precisely captures the function at fault. We symbolically simplify the term $e \triangleright t$ aiming to simplify `BAD`s away. In case there is any `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 $e \triangleright t$ and prove a theorem "$e \triangleright t$ is crash-free $\Longleftrightarrow\ e \in t$" (§4). The "crash-free" means "`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 `BAD` statically (§5). 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 *logicization* technique that transforms expressions to logical formulae, inspired by [19, 20] and axiomatization of functions that interactive theorem provers perform before calling SMT sovlers. However, we have to deal with non-total terms and that is the key contribution of the *logicization* (§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:

```
contract inc = {x | x > 0} -> {y | y > 0}
let inc = fun v -> v + 1
let t1 = inc 0
```

A dynamic checker wraps the `inc` in `t1` with its contract $t_{\texttt{inc}}$:

$$\texttt{let t1 = }(\texttt{inc} \overset{\texttt{BAD}^{l}}{\underset{\texttt{BAD}^{l'}}{\bowtie}} t_{\texttt{inc}})\ 0$$

where l is $(2, 5, \text{“}\texttt{inc}\text{”})$ indicating the source location where `inc` is defined (row:2,col:5) and l' is $(3, 10, \text{“}\texttt{t1}\text{”})$ indicating the location of the call site with caller's name. This wrapped `t1` expands to:

$$\begin{array}{l}(\lambda x_1.\ \texttt{let}\ y =\ \texttt{inc}\ \boxed{\begin{array}{l}(\texttt{let}\ x = x_1\ \texttt{in}\\ \ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{BAD}^{(3,10,\text{“}\texttt{t1}\text{”})})\end{array}}\\ \qquad \texttt{in}\ \boxed{\texttt{if}\ y > 0\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}^{(2,5,\text{“}\texttt{inc}\text{”})}}\ )\ 0\end{array}$$

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, `t1` calls `inc` with 0, which fails `inc`'s precondition. Running the above wrapped code, we get $\texttt{BAD}^{(3,10,\text{“}\texttt{t1}\text{”})}$, which precisely blames `t1`.

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:

$$f \overset{\texttt{BAD}^{f}}{\underset{\texttt{BAD}^{?}}{\bowtie}} t$$

which behaves the same as f except that (a) if f disobeys t, it blames f, signaled by $\texttt{BAD}^{f}$; (b) if the context uses f in a way not permitted by t, it blames the caller of f, signaled by $\texttt{BAD}^{?}$ 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:

$$\text{splitting } \overset{\texttt{BAD}^{f}}{\underset{\texttt{BAD}^{?}}{\bowtie}} \text{ into half: } e \rhd t = e \overset{\texttt{BAD}^{f}}{\underset{\texttt{UNR}^{?}}{\bowtie}} t \text{ and } e \lhd t = e \overset{\texttt{UNR}^{f}}{\underset{\texttt{BAD}^{?}}{\bowtie}} t.$$

The $\rhd$ ("ensures") and the $\lhd$ ("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, $(\lambda v.v + 1) \rhd t_{\texttt{inc}}$ expands to:

$$\begin{array}{l}\lambda x_1.\ \texttt{let}\ y =\ (\lambda v.v + 1)\\ \qquad\qquad (\texttt{let}\ x = x_1\ \texttt{in}\ \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{UNR}^{?})\ \texttt{in}\\ \quad \texttt{if}\ y > 0\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}^{(2,5,\text{“}\texttt{inc}\text{”})}\end{array}$$

At the definition site of a function, $f = e$, we assume f's precondition holds and assert its postcondition. If all `BAD`s in $e \rhd t$ 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:

$$\begin{array}{l}\lambda x_1.\ \texttt{let}\ y = (\lambda v.v + 1)\ (\texttt{if}\ x_1 > 0\ \texttt{then}\ x_1\ \texttt{else}\ \texttt{UNR}^{?})\ \texttt{in}\\ \quad \texttt{if}\ y > 0\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}^{(2,5,\text{“}\texttt{inc}\text{”})}\end{array}$$

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:

$\lambda v.v + 1$	$\exists x_2, (\forall v, x_2(v) = v + 1)$
$\texttt{if}\ x_1 > 0\ \texttt{then}\ x_1\ \texttt{else}\ \texttt{UNR}^{?}$	$\exists x_3, (x_1 > 0 \Rightarrow x_3 = x_1) \vee$ $(not(x_1 > 0) \Rightarrow \mathit{false})$

One can think of the existentially quantified x_2 (and x_3) denoting the expression itself. For the RHS of y, we have logical formula:

$$\begin{array}{l}\forall y, \exists x_2, (\forall v, x_2(v) = v + 1) \wedge (\exists x_3, (x_1 > 0 \Rightarrow x_3 = x_1)\\ \wedge (not(x_1 > 0) \Rightarrow \mathit{false}) \wedge y = x_2(x_3)) \qquad [\text{Q1}]\end{array}$$

We check the validity of $\forall x_1, \text{Q1} \Rightarrow y > 0$ by consulting an SMT solver. As $\forall x_1, \text{Q1} \Rightarrow y > 0$ is valid, we know the $\texttt{BAD}^{(2,5,\text{“}\texttt{inc}\text{”})}$ is not reachable, thus `inc` satisfies its contract.

Consider the function `f1` and its contract $t_{\texttt{f1}}$ in §1. So $\texttt{f1} \rhd t_{\texttt{f1}}$ is $(\lambda g.(g\ 1) - 1) \rhd (\{x \mid x \geq 0\} \rightarrow \{y \mid y \geq 0\}) \rightarrow \{z \mid z \geq 0\}$, which expands to:

$$\begin{array}{l}\lambda x_1.\ \texttt{let}\ z =\ (\lambda g.(g\ 1) - 1)\\ \qquad\qquad (\lambda x_2.\ \texttt{let}\ y = x_1\ (\ \texttt{let}\ x = x_2\ \texttt{in}\\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{if}\ x \geq 0\ \texttt{then}\ x\\ \qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ \texttt{BAD}^{(4,5,\text{“}f_1\text{”})})\ \texttt{in}\\ \qquad\qquad\qquad \texttt{if}\ y \geq 0\ \texttt{then}\ y\ \texttt{else}\ \texttt{UNR}^{?})\ \texttt{in}\\ \quad \texttt{if}\ z \geq 0\ \texttt{then}\ z\ \texttt{else}\ \texttt{BAD}^{(4,5,\text{“}f_1\text{”})}\end{array}$$

After applying some conventional simplification rules, we have:

$$\begin{array}{l}\text{R1}:\ \ \lambda x_1.\ \texttt{let}\ z =\ \texttt{let}\ y = x_1\ 1\ \texttt{in}\\ \qquad\qquad\qquad\qquad \texttt{if}\ y \geq 0\ \texttt{then}\ y - 1\ \texttt{else}\ \texttt{UNR}^{?}\\ \qquad\qquad \texttt{if}\ z \geq 0\ \texttt{then}\ z\ \texttt{else}\ \texttt{BAD}^{(4,5,\text{“}f_1\text{”})}\end{array}$$

We see that the inner $\texttt{BAD}^{(4,5,\text{“}f_1\text{”})}$ has been simplified away, because $x = x_2 = 1$ and ($\texttt{if}\ 1 \geq 0\ \texttt{then}\ 1\ \texttt{else}\ \texttt{BAD}^{(4,5,\text{“}f_1\text{”})}$) is simplified to 1. As we cannot prove $\forall x_1, \forall z, (\exists y, y = x_1\ 1 \wedge (y \geq 0 \Rightarrow z = y - 1)) \Rightarrow z \geq 0$ to be valid, the other $\texttt{BAD}^{(4,5,\text{“}f_1\text{”})}$ 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, $\texttt{f1} \rhd t_{\texttt{f1}}$ checks whether `f1` satisfies its postcondition by assuming its precondition holds. At each call site of `f1`, we wrap the function with $\lhd$. 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 $\{x \mid \texttt{true}\}$. Say zut $\lhd \{x \mid \texttt{true}\} = $ zut, we then have the term $\texttt{f3} \rhd t_{\texttt{f3}}$ to be:

$$((\texttt{f1} \lhd t_{\texttt{f1}})\ \texttt{zut}) \rhd \{v \mid v > 0\}$$

which *requires* f3 to satisfy f1's precondition and assumes f1 satisfies its postcondition because $\texttt{f1} \rhd t_{\texttt{f1}}$ 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 $f \lhd t$. The $\texttt{f3} \rhd t_{\texttt{f3}}$ expands to:

$$\begin{array}{l}
\texttt{let}\ v = \ \texttt{let}\ z = \ \texttt{f1} \\
\qquad\qquad\qquad (\lambda x_2.\texttt{let}\ y = \texttt{zut}\ \ (\texttt{let}\ x = x_2\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{if}\ x \geq 0\ \texttt{then}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else UNR}^{(7,10,\text{“}f_1\text{”})})\ \texttt{in} \\
\qquad\qquad\qquad\ \ \texttt{if}\ y \geq 0\ \texttt{then}\ y\ \texttt{else BAD}^{(7,10,\text{“}f_3\text{”})})\ \texttt{in} \\
\qquad\quad \texttt{if}\ z \geq 0\ \texttt{then}\ z\ \texttt{else UNR}^{(7,10,\text{“}f_1\text{”})} \\
\texttt{if}\ v \geq 0\ \texttt{then}\ v\ \texttt{else BAD}^{(7,10,\text{“}f_3\text{”})}
\end{array}$$

As $\lhd$ is dual to $\rhd$, the RHS of v is actually a copy of the earlier $\texttt{f1} \rhd t_{\texttt{f1}}$ 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 $\texttt{UNR}^?$ becomes $\texttt{BAD}^{(7,10,\text{“}f_3\text{”})}$ and the $\texttt{BAD}^{(4,5,\text{“}f_1\text{”})}$ becomes $\texttt{UNR}^{(7,10,\text{“}f_1\text{”})}$. At definition site where the caller is unknown, we use the location of f1, i.e. $(4, 5, \text{“}f_1\text{”})$. Once its caller is known, we use $(7, 10, \text{“}f_1\text{”})$. 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 $\forall v.(\exists z.z \geq 0 \wedge v = z) \Rightarrow v \geq 0$, the $\texttt{f3} \rhd t_{\texttt{f3}}$ can be simplified to (say R2):

$$\begin{array}{l}
\texttt{let}\ z = \ \texttt{f1} \\
\qquad (\lambda x_2.\ \texttt{let}\ y = \texttt{zut}\ \ (\texttt{let}\ x = x_2\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad \texttt{if}\ x > 0\ \texttt{then}\ x \\
\qquad\qquad\qquad\qquad\qquad \texttt{else UNR}^{(7,10,\text{“}f_1\text{”})})\ \texttt{in} \\
\qquad\quad \texttt{if}\ y \geq 0\ \texttt{then}\ y\ \texttt{else BAD}^{(7,10,\text{“}f_3\text{”})})\ \texttt{in} \\
\texttt{if}\ z \geq 0\ \texttt{then}\ z\ \texttt{else UNR}^{(7,10,\text{“}f_1\text{”})}
\end{array}$$

One BAD remains. We can either report this potential contract violation at compile-time or continue a DCC. For SCC, we have checked $\texttt{f1} \rhd t_{\texttt{f1}}$, but for DCC, to invoke $\texttt{f1} \rhd t_{\texttt{f1}}$, 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:

$$(\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else UNR}) \Longrightarrow e_1 \qquad [\texttt{rmUNR}]$$

(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:

$$\begin{array}{l}
\texttt{f1}\sharp = \lambda x_1.\ \texttt{let}\ z = (\texttt{let}\ y = (x_1\ 1)\ \texttt{in}\ y - 1)\ \texttt{in} \\
\qquad\qquad\quad \texttt{if}\ z \geq 0\ \texttt{then}\ z\ \texttt{else BAD}^{(4,5,\text{“}f_1\text{”})} \\
\texttt{f2}\sharp = \ \texttt{f1}\sharp\ \ (\lambda x_2.\texttt{let}\ y = \texttt{zut}\ x_2\ \texttt{in} \\
\qquad\qquad\quad \texttt{if}\ y \geq 0\ \texttt{then}\ y\ \texttt{else BAD}^{(7,10,\text{“}f_3\text{”})})
\end{array}$$

respectively, which is the *residual* code being run. We show in §6 that HCC blames a function f_i iff DCC blames f_i.

Summary Given a definition $f = e$ and a contract t, to check e satisfies t (written $e \in t$), we perform these steps. (1) Construct $e \rhd t$. (2) Simplify $e \rhd t$ 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 $f\sharp$ by simplifying e' with the rule [rmUNR], and run the program with each f being replaced by $f\sharp$.

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

$$\begin{array}{llll}
x, f \in \textbf{Variables} & & T \in \textbf{Type constructors} & \\
 & & K \in \textbf{Data constructors} & \\
 & & & \\
pgm & ::= & def_1, \ldots, def_n & \textbf{Program} \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \overrightarrow{\tau}\ T \mid \tau_1 \to \tau_2 & \textbf{Types} \\
 & & & \\
t & \in & \textbf{Contracts} & \\
t & ::= & \{x \mid p\} & \text{predicate contract} \\
 & \mid & x\colon t_1 \to t_2 & \text{dependent function contract} \\
 & \mid & (x\colon t_1, t_2) & \text{dependent tuple contract} \\
 & \mid & \texttt{Any} & \text{polymorphic Any contract} \\
 & & & \\
def & \in & \textbf{Definitions} & \\
def & ::= & \texttt{type}\ \overrightarrow{'\alpha}\ T\ =\ \overrightarrow{K\ \texttt{of}\ \tau} & \\
 & \mid & \texttt{contract}\ f = t & \\
 & \mid & \texttt{let}\ f\ \overrightarrow{x} = e & \text{top-level function} \\
 & \mid & \texttt{let rec}\ f\ \overrightarrow{x} = e & \text{top-level recursive function} \\
 & & & \\
a, e, p & \in & \textbf{Exp} & \textbf{Expressions} \\
a, e, p & ::= & n & \text{integers} \\
 & \mid & r & \text{blame} \\
 & \mid & x \mid \lambda(x^\tau).e \mid e_1\ e_2 & \\
 & \mid & \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{alt} & \text{pattern-matching} \\
 & \mid & K\ \overrightarrow{e} & \text{constructor} \\
 & & & \\
alt & ::= & K\ (x_1^{\tau_1}, \ldots, x_n^{\tau_n})\ \to\ e & \textbf{Alternatives} \\
r & ::= & \texttt{BAD}^l\ \mid\ \texttt{UNR}^l & \textbf{Blames} \\
l & ::= & (n_1, n_2, \texttt{String}) & \textbf{Label} \\
 & & & \\
val & ::= & n \mid x \mid r \mid K\ \overrightarrow{val} \mid \lambda(x^\tau).e & \textbf{Values} \\
tv & ::= & n \mid x \mid K\ \overrightarrow{tv} & \\
tval & ::= & tv \mid \lambda(x^\tau).e & \textbf{Trivial values}
\end{array}$$

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. Base 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 (e_1, e_2) with type ('a,'b) product = Pair of 'a * 'b. A local let-expression let $x = e_1$ in e_2 is a syntactic sugar for $(\lambda x.e_2)\ e_1$. An if-expression if e_0 then e_1 else e_2 is syntactic sugar for match e_0 with $\{\texttt{true} \to e_1; \texttt{false} \to e_2\}$.

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^τ or t^τ) whenever necessary. Type checking material is omitted, but can be found in [40].

The two contract exceptions (also called blames) $\mathtt{BAD}^l$ and $\mathtt{UNR}^l$ 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::l -> x
```

where `raise` : $\forall\alpha.$ `Exception` $\rightarrow \alpha$. The `Exception` is a built-in data type for exceptions and `Emptylist` has type `Exception`. As we do not have `try-with` in language M (leaving it as future work), a preprocessing converts `raise Emptylist` to $\mathtt{BAD}^{\mathtt{head}}$.

We have four forms of contracts. The p in a predicate contract $\{x \mid p\}$ 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, $x{:}\{y \mid y > 0\} \rightarrow \{z \mid z > x\}$ says that, the input is greater than 0 and the output is greater than the input. We can use a shorthand $\{x \mid x > 0\} \rightarrow \{z \mid z > x\}$ by assuming x scopes over the RHS of $\rightarrow$. The $\rightarrow$ is right associative. Similarly, dependent tuple contracts allow us to describe dependency between two components of a tuple. For example, $(x{:}\{y \mid y > 0\}, \{z \mid z > x\})$ has short hand $(\{x \mid x > 0\}, \{z \mid z > x\})$. Contract `Any` is a universal contract that any expression satisfies. We support higher order contracts, e.g. $k{:}\ (\{x \mid x > 0\} \rightarrow \{y \mid y > x\}) \rightarrow \{z \mid k\ 5 > z\}$ for a function `let f g = g 2`.

3.2 Operational semantics

The semantics of our language is given by reduction rules in Figure 2. For a top-level function, we fetch its definition from the evaluation environment Δ. We adapt some basic definitions from [41]. Definition 1 defines the usual contextual equivalence. Two expressions are said to be semantically equivalent, if and only if under all (closing) contexts, if one evaluates to a blame r, the other also evaluates to the same r.

Definition 1 (Semantically Equivalent)**.** *Two expressions e_1 and e_2 are semantically equivalent, namely $e_1 \equiv_s e_2$, iff for all closing $\mathcal{C}$, for all r,* $\quad \mathcal{C}[\![e_1]\!] \rightarrow^* r \quad \Longleftrightarrow \quad \mathcal{C}[\![e_2]\!] \rightarrow^* r$

$$\frac{\mathtt{let}\ (\mathtt{rec})\ f = e \in \Delta}{f \rightarrow e} \quad \text{[E-top]}$$

$$(\lambda x.e)\ val \rightarrow e[val/x] \quad \text{[E-beta]}$$

$$\mathtt{match}\ K\ \overrightarrow{val}\ \mathtt{with}\ \overrightarrow{K\ \vec{x} \rightarrow e} \rightarrow e[\overrightarrow{val/x}] \quad \text{[E-match]}$$

$$\frac{e_1 \rightarrow e_2}{\mathcal{C}[\![e_1]\!] \rightarrow \mathcal{C}[\![e_2]\!]} \quad \text{[E-ctx]} \qquad \mathcal{C}[\![r]\!] \rightarrow r \quad \text{[E-exn]}$$

$$\begin{array}{lll}\text{Contexts } \mathcal{C} & ::= & [\![\bullet]\!] \mid \mathcal{C}\ e \mid val\ \mathcal{C} \mid K\ \overrightarrow{val}\ \mathcal{C}\ \vec{e} \\ & \mid & \mathtt{match}\ \mathcal{C}\ \mathtt{with}\ \overrightarrow{alt}\end{array}$$

Figure 2: Semantics of the language M

We use `BAD` to signal that something has gone wrong in a program, which can be a program failure or a contract violation.

Definition 2 (Crash)**.** *A closed term e* crashes *iff $e \rightarrow^*$ `BAD`.*

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

Definition 3 (Diverges)**.** *A closed expression e* diverges, *written $e\uparrow$, iff either $e \rightarrow^*$ `UNR`, or there is no value val such that $e \rightarrow^* val$.*

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

Definition 4 (Syntactic safety)**.** *A (possibly-open) expression e is* syntactically safe *iff `BAD` $\notin_s e$. Similarly, a context $\mathcal{C}$ is syntactically safe iff `BAD` $\notin_s \mathcal{C}$.*

The notation `BAD` $\notin_s e$ means `BAD` does not syntactically appear anywhere in e, similarly for `BAD` $\notin_s \mathcal{C}$. For example, $\lambda x.x$ is syntactically safe while $\lambda x.\ (\mathtt{BAD}, x)$ is not.

Definition 5 (Crash-free Expression)**.** *A (possibly-open) expression e is crash-free iff :*

$$\forall\mathcal{C}.\ \mathtt{BAD} \notin_s \mathcal{C} \text{ and } (\mathcal{C}[\![e]\!])^{\mathtt{bool}} \ \Rightarrow\ \mathcal{C}[\![e]\!] \not\rightarrow^* \mathtt{BAD}$$

The notation $(\mathcal{C}[\![e]\!])^{\mathtt{bool}}$ means $\mathcal{C}[\![e]\!]$ is closed and well-typed. The quantified context $\mathcal{C}$ 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. $\lambda x.\mathtt{if}\ x * x \geq 0\ \mathtt{then}\ x + 1\ \mathtt{else}\ \mathtt{BAD}$.

Lemma 1 (Syntactically safe expression is crash-free)**.**

$$e \textit{ is syntactically safe} \quad \Rightarrow \quad e \textit{ is crash-free}$$

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 $\mathtt{BAD}^l$ and $\mathtt{UNR}^l$ are closed expressions even if l is not explicitly bound.

4. 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.

Definition 6 (Total contract)**.** *A contract t is total iff*

$$\begin{array}{ll} & t \textit{ is } \{x \mid p\} \textit{ and } \lambda x.p \textit{ is total (i.e. crash-free, terminating)} \\ \textit{or} & t \textit{ is } x{:}\ t_1 \rightarrow t_2 \textit{ and } t_1 \textit{ is total and} \\ & \qquad \textit{for all } val_1 \in t_1, t_2[val_1/x] \textit{ is total} \\ \textit{or} & t \textit{ is } (x{:}t_1, t_2) \textit{ and } t_1 \textit{ is total and} \\ & \qquad \textit{for all } val_1 \in t_1, t_2[val_1/x] \textit{ is total} \\ \textit{or} & t \textit{ is } \mathtt{Any}\end{array}$$

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 $\lambda x.p$ is a terminating function. For example, contract $\{x \mid x \neq \mathtt{[]}\} \rightarrow \{y \mid \mathtt{head}\ x > y\}$ is total in our framework because $\mathtt{head}\ x$ does not crash for all x satisfying $\{x \mid x \neq \mathtt{[]}\}$. Such a contract is not total in [7] because a crashing function `head` is called in a predicate contract.

4.1 A semantics for contract satisfaction

We give the semantics of contracts by defining "e satisfies t" (written $e \in t$) 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:

$$
\begin{array}{rcll}
e \in \{x \mid p\} & \Longleftrightarrow & e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \to^* \mathtt{true}) & \text{[A1]} \\
e \in x{:}\, t_1 \to t_2 & \Longleftrightarrow & e\uparrow \text{ or } (e \to^* \lambda x.e_2 \text{ and } \forall val_1 \in t_1.\ (e\ val_1) \in t_2[val_1/x]) & \text{[A2]} \\
e \in (x{:}\, t_1, t_2) & \Longleftrightarrow & e\uparrow \text{ or } (e \to^* (val_1, val_2) \text{ and } val_1 \in t_1 \text{ and } val_2 \in t_2[val_1/x]) & \text{[A3]} \\
e \in \mathtt{Any} & \Longleftrightarrow & \mathtt{true} & \text{[A4]}
\end{array}
$$

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 $t_1 \leqq t_2$, iff $\forall e.\ e \in t_1 \Rightarrow e \in t_2$*

It is obvious that 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 Any for a strict language. Consider:

```
contract fail = Any
let fail = raise Error
```

where Error has type Exception. One can think of Any as $\forall\alpha.\alpha$. In [7] and other refinement type checking framework [5, 25, 37], they give function like fail a function contract $\{x \mid \mathtt{false}\} \to \{x \mid \mathtt{true}\}$ so that the precondition $\{x \mid \mathtt{false}\}$ allows their system to blame all the callers of fail. This is somewhat ad hoc. More discussion on the contract 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 $\triangleright^{l_1}_{l_2}$ and $\triangleleft^{l_1}_{l_2}$ respectively. The wrapped expression $e \overset{r_1}{\underset{r_2}{\bowtie}} t$ 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.

$$
e \triangleright t = e \overset{\mathtt{BAD}^{l_1}}{\underset{\mathtt{UNR}^{l_2}}{\bowtie}} t \qquad e \triangleleft t = e \overset{\mathtt{UNR}^{l_2}}{\underset{\mathtt{BAD}^{l_1}}{\bowtie}} t
$$

$$
e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \ = \ \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{if}\ p\ \mathtt{then}\ x\ \mathtt{else}\ r_1 \qquad \text{[P1]}
$$

$$
\begin{array}{ll}
e \overset{r_1}{\underset{r_2}{\bowtie}} x{:}t_1 \to t_2 = \mathtt{let}\ y = e\ \mathtt{in} & \text{[P2]} \\
\qquad \lambda x_1.((y\ (x_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_2}{\underset{\underline{r_1}}{\bowtie}} t_1)/x]) &
\end{array}
$$

$$
\begin{array}{ll}
e \overset{r_1}{\underset{r_2}{\bowtie}} (x{:}t_1, t_2) = \mathtt{match}\ e\ \mathtt{with} & \text{[P3]} \\
\qquad (x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1, x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_2}{\underset{\underline{r_1}}{\bowtie}} t_1)/x]) &
\end{array}
$$

$$
e \overset{r_1}{\underset{r_2}{\bowtie}} \mathtt{Any} = r_2 \qquad \text{[P4]}
$$

Figure 4: Contract checking with the wrappers

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^τ, closed and total contract t^τ,*

$$(e \triangleright t) \textit{ is crash-free} \quad \Longleftrightarrow \quad e \in t$$

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

Theorem 2 (Soundness of contract checking)**.** *For all closed expression e^τ, closed and terminating contract t^τ,*

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

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 $t_2[(v \overset{r_2}{\underset{\underline{r_1}}{\bowtie}} t_1/x]$ in [P2] and [P3] says that, we wrap each (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 | x > 0 } -> {y | y > 0 })
             -> {z | k 0 > -1}
let f g = g 2
let t2 = f (fun x -> x)
```

a contract violation occurs in {z | k 0 > -1} because the call k 0 fails k's precondition {x | 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, we use $\underline{r_1}$ to indicate that the label of r_1 is replaced by the name of the contract. In [7], they use an ad hoc fix, i.e. using UNR instead of $\underline{r_1}$ 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 Γ is an environment mapping a variable to its contract, we check $e[(f_i \triangleleft t_{f_i})/f_i] \triangleright t[(f_i \triangleleft t_{f_i})/f_i]$ where f_i are free variables in e (or t) and are in the domain of Γ. Note that f_i can also be a recursive call f and $f \triangleleft t$ in $e[(f \triangleleft t)/f] \triangleright t$ is like an induction hypothesis. As $e[(f_i \triangleleft t_{f_i})/f_i] \triangleright t[(f_i \triangleleft t_{f_i})/f_i]$ 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 loop x = loop x
contract fb = {x | loop x} -> {y | true}
let fb x = head []
```

$\mathtt{fb} \triangleright t_{\mathtt{fb}}$ is $\lambda x_1.((\lambda x.\mathtt{head}\ [\,])\ (\mathtt{if\ loop}\ x_1\ \mathtt{then}\ x_1\ \mathtt{else\ BAD}))$, 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 $\bowtie$ (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

$$
\begin{array}{rcll}
\langle \mathcal{H} \mid n \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid n \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[S-const]} \\
\langle \mathcal{H} \mid r \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid r \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[S-exn]} \\
\langle \mathcal{H}[x \mapsto tval] \mid x \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle\!\langle \mathcal{H}[x \mapsto tval] \mid tval \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[S-var1]} \\
\text{If } x \notin \mathcal{H}, \quad \langle \mathcal{H} \mid x \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid x \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[S-var2]} \\
\langle \mathcal{H} \mid \lambda x^{\tau}.e \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid e \mid (\lambda x.\bullet) :: S \mid \mathcal{L}, \forall x : [\![\tau]\!] \rangle & \text{[S-lam]} \\
\langle \mathcal{H} \mid e_1\ e_2 \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_1 \mid (\bullet\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-app]} \\
\langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ alts \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_0 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-match]} \\
\langle \mathcal{H} \mid K\ (a_1, \ldots, e_i, \ldots, e_n) \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_i \mid (K\ (a_1, \ldots, \bullet, \ldots, e_n)]) :: \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-K]} \\
\text{if } x \notin fv(e), \quad \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid (\bullet\ e) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ e \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-letL]} \\
\text{if } fv(e) \cap \overrightarrow{x_i} = \emptyset, \quad \langle \mathcal{H} \mid \begin{array}{l}(\texttt{match}\ e_0\ \texttt{with} \\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e_i})\end{array} \mid (\bullet\ e) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e_i\ e} \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-matchL]} \\
\text{if } x \notin fv(a), \quad \langle \mathcal{H} \mid val \mid (\bullet\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ val\ e_2 \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-letR]} \\
\text{if } fv(val) \cap \overrightarrow{x} = \emptyset, & & & \\
\langle \mathcal{H} \mid val \mid (\bullet\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e})) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow val\ e} \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-matchR]} \\
\text{if } fv(alts) \cap \overrightarrow{x} = \emptyset, & & & \\
\langle \mathcal{H} \mid \begin{array}{l}\texttt{match}\ e_0\ \texttt{with} \\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e}\end{array} \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \begin{array}{l}\texttt{match}\ e_0\ \texttt{with} \\ \overrightarrow{K\ \overrightarrow{x} \rightarrow \texttt{match}\ e\ \texttt{with}\ alts}\end{array} \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-match-match]} \\
\text{if } x \notin fv(alts), & & & \\
\langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-match-let]} \\
& & & \\
\langle\!\langle \mathcal{H} \mid a \mid [\,] \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & a & \text{[R-done]} \\
\text{if } (s \neq \texttt{match}\ e\ \texttt{with}\ K\ \overrightarrow{x} \rightarrow (\bullet, \mathcal{S}, \mathcal{L})), \ \langle\!\langle \mathcal{H} \mid r \mid s :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid r \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[R-r]} \\
\langle\!\langle \mathcal{H} \mid a \mid (\lambda x.\bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid \lambda x.a \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[R-lam]} \\
& & & \\
\boxed{\text{Rules below: } a \notin \{\texttt{BAD}^l, \texttt{UNR}^l\} \text{ by default}} & & & \\
& & & \\
\langle\!\langle \mathcal{H} \mid a \mid (\bullet\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_2 \mid (a\ \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle & \text{[R-fun]} \\
\langle\!\langle \mathcal{H} \mid tval \mid ((\lambda x.a_1)\ \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle\!\langle \mathcal{H}[x \mapsto tval] \mid a_1 \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[R-beta]} \\
\text{if } a_1 \neq \lambda x.a' \text{ or } a \neq tval, \quad \langle\!\langle \mathcal{H} \mid a \mid (a_1\ \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid a_1\ a \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[R-app]} \\
\langle\!\langle \mathcal{H} \mid a_n \mid (K\ a_1 \ldots \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid K\ \overrightarrow{a} \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[R-K]} \\
\langle\!\langle \mathcal{H} \mid K\ \overrightarrow{a} \mid (\texttt{match}\ \bullet\ \texttt{with}\ \{\ldots; K\ \overrightarrow{x} \rightarrow e; \ldots\}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid \texttt{let}\ \overrightarrow{x = a}\ \texttt{in}\ e \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[R-K-match]} \\
\text{if exists } (K\ \overrightarrow{x}) \text{ such that } \mathcal{L} \Rightarrow (\exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})}), & & & \\
\langle\!\langle \mathcal{H} \mid a \mid (\texttt{match}\ \bullet\ \texttt{with}\ \overrightarrow{K\ x^{\not\tau}} \rightarrow e) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L}, \forall \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})} \rangle & \text{[R-s-match]} \\
\text{if for all } (K\ \overrightarrow{x}) \text{ such that } \mathcal{L} \not\Rightarrow (\exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})}), & & & \\
\langle\!\langle \mathcal{H} \mid a \mid (\texttt{match}\ \bullet\ \texttt{with}\ \overrightarrow{K\ x^{\not\tau}} \rightarrow e) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \overrightarrow{\langle \mathcal{H} \mid e \mid \begin{array}{l}(\texttt{match}\ a\ \texttt{with}\ K\ \overrightarrow{x^{\not\tau}} \\ \rightarrow (\bullet, \mathcal{S}, \mathcal{L})) :: [\,]\end{array} \mid \begin{array}{l}\mathcal{L}, \forall \overrightarrow{x : [\![\tau]\!]}, \\ [\![a]\!]_{(K\ \overrightarrow{x})}\end{array} \rangle} & \text{[R-s-save]} \\
\overrightarrow{\langle\!\langle \mathcal{H} \mid a \mid (\texttt{match}\ a_0\ \texttt{with}\ K\ \overrightarrow{x} \rightarrow (\bullet, \mathcal{S}, \mathcal{L})) :: \mathcal{S}' \mid \mathcal{L}' \rangle\!\rangle} & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid \texttt{match}\ a_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow a} \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \text{[R-match]} \\
\text{for some } \mathcal{S}' \text{ and } \mathcal{L}' \text{ and } a \text{ can be } r & & & \\
\langle\!\langle \mathcal{H} \mid a \mid (\texttt{let}\ x^{\tau} = \bullet\ \texttt{in}\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_2 \mid (\texttt{let}\ x = a\ \texttt{in}\ \bullet) :: \mathcal{S} \mid \mathcal{L}, \forall x\colon [\![\tau]\!], [\![a]\!]_x \rangle & \text{[R-let-save]}
\end{array}
$$

Figure 5: SL machine

$$
\begin{array}{rcll}
(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ e & \Longrightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ e & \text{[letL]} \\
\text{if } fv(e) \cap \overrightarrow{x} = \emptyset, \quad (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e_i})\ e & \Longrightarrow & \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow (e_i\ e)} & \text{[matchL]} \\
\text{if } x \notin fv(e), \quad val\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & \Longrightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ val\ e_2 & \text{[letR]} \\
\text{if } fv(val) \notin \overrightarrow{x}, \quad val\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e}) & \Longrightarrow & \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow val\ e} & \text{[matchR]} \\
\text{if } fv(alts) \cap \overrightarrow{x} = \emptyset, & & & \\
\texttt{match}\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow e})\ \texttt{with}\ alts & \Longrightarrow & \texttt{match}\ e_o\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow \texttt{match}\ e\ \texttt{with}\ alts} & \text{[match-match]} \\
\text{if } x \notin fv(alts), \quad \texttt{match}\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ \texttt{with}\ alts & \Longrightarrow & \texttt{let}\ x = e_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts & \text{[match-let]} \\
\texttt{match}\ K\ a_1 \ldots a_n\ \texttt{with}\ \{\ldots; K\ x_1 \ldots x_n \rightarrow e; \ldots\} & \Longrightarrow & \texttt{let}\ x_1 = a_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = a_n\ \texttt{in}\ e & \text{[K-match]}
\end{array}
$$

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 $\langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L} \rangle$

is pronounced *simplify* and a 4-tuple $\langle\!\langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ is pronounced *rebuild* where

- $\mathcal{H}$ is an environment mapping variables to trivial values;
- e is the expression under simplification (or being rebuilt);
- $\mathcal{S}$ is a stack which embodies the simplification context, or continuation that will consume a simplified expression;
- $\mathcal{L}$ is a logical store which contains the ctx-info in logical formula form; its syntax is

$$\mathcal{L} ::= \emptyset \mid \forall x : \tau, \mathcal{L} \mid \phi, \mathcal{L}$$

where ϕ 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 $\langle \emptyset \mid e \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* a$.*

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 $\langle \emptyset \mid e \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* a$, then $e \equiv_s a$.*

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 $\langle \mathcal{H} \mid e_1 \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow^* \langle \mathcal{H} \mid e_3 \mid \mathcal{S} \mid \mathcal{L} \rangle$ and $\langle \mathcal{H} \mid e_2 \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow^* \langle \mathcal{H} \mid e_3 \mid \mathcal{S} \mid \mathcal{L} \rangle$, then $e_1 \equiv_s e_2$. (See Definition 1 for $\equiv_s$.)

Theorem 5 (Soundness of static contract checking)**.** *For all closed expression e, and closed and terminating contract t,*

$$\langle \emptyset \mid e \triangleright t \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* e' \text{ and } \mathtt{BAD} \notin_s e' \quad \Rightarrow \quad e \in t$$

Proof. By Theorem 4, Lemma 1 and Theorem 2. □

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) as we do hybrid contract checking, we want to send the residual code with undischarged blames 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 $\mathcal{H}$ if it denotes a trivial value. A variable denoting a top-level function is not put in $\mathcal{H}$. Variables in $\mathcal{H}$ are inlined by [S-var1] while variables not in $\mathcal{H}$ 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 in $\mathcal{S}$ is

$$\begin{array}{rl} s ::= & [\,] \mid (\bullet\ e) :: s \mid (e\ \bullet) :: s \mid (\lambda x.\bullet) :: s \mid \mathtt{let}\ x = \bullet\ \mathtt{in}\ e \\ & \mid (\mathtt{match}\ \bullet\ \mathtt{with}\ alt) :: s \mid \mathtt{let}\ x = e\ \mathtt{in}\ \bullet \\ & \mid (\mathtt{match}\ e_0\ \mathtt{with}\ \overrightarrow{K\ \vec{x} \to (\bullet, \mathcal{S}, \mathcal{L})}) :: s \end{array}$$

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, $\vec{x}$ 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

$$\mathtt{match}\ \mathtt{Some}\ e\ \mathtt{with}\ \{\mathtt{Some}\ x \to 5; \mathtt{None}\ \to \mathtt{BAD}\}$$

(where e is a crash-free expression, not a value) to $\mathtt{let}\ x = e\ \mathtt{in}\ 5$ 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 [R-r] 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 $\mathcal{S}$, we use [R-beta]; together with [S-var1], they implement a beta-reduction [E-beta]. Bound variables are renamed when necessary.

The logical store $\mathcal{L}$ 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:

$$\langle \mathcal{H} \mid \begin{array}{l} (\lambda x.\ \ \mathtt{if}\ x > 0\ \mathtt{then}\ (\mathtt{if}\ x+1 > 0 \\ \qquad\qquad\qquad\qquad\quad \mathtt{then}\ 5\ \mathtt{else}\ \mathtt{BAD}) \\ \qquad\quad \mathtt{else}\ \mathtt{UNR}) \end{array} \mid [\,] \mid \emptyset \rangle$$

The [S-lam] puts $\forall x : \mathtt{int}$ in $\mathcal{L}$, which is initially empty:

$$\langle \mathcal{H} \mid \begin{array}{l} (\mathtt{if}\ x > 0\ \mathtt{then} \\ (\mathtt{if}\ x+1 > 0 \\ \mathtt{then}\ 5\ \mathtt{else}\ \mathtt{BAD})\ \mathtt{else}\ \mathtt{UNR}) \end{array} \mid (\lambda x.\bullet) :: [\,] \mid \forall x : \mathtt{int} \rangle$$

The [S-match] starts to simplify the scrutinee $x > 0$, which is being rebuilt after a few trivial steps.

$$\langle\!\langle \mathcal{H} \mid x > 0 \mid \begin{array}{l} (\mathtt{if}\ \bullet\ \mathtt{then}\ \ (\mathtt{if}\ x+1 > 0 \\ \qquad\qquad\qquad\ \mathtt{then}\ 5\ \mathtt{else}\ \mathtt{BAD}) \\ \mathtt{else}\ \mathtt{UNR}) :: (\lambda x.\bullet) :: [\,] \end{array} \mid \forall x : \mathtt{int} \rangle\!\rangle$$

Before applying the transition [R-s-save], we check whether $x > 0$ or $not(x > 0)$ is implied by $\mathcal{L}$ to see whether the transition [R-s-match] can be applied. The transition [R-s-match] implements [E-match], where the side condition "if $\exists(K\ \vec{x}),\ \mathcal{L} \Rightarrow [\![a]\!]_{(K\ \vec{x})}$" checks if there is any branch $K\ \vec{x}$ that matches the scrutinee a. But the current information in $\mathcal{L}$ 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 $[\![a]\!]_{(K\ \vec{x})}$ (explained later) and put it in $\mathcal{L}$ and simplify both branches. Note that we put $x > 0$ in $\mathcal{L}$ for the `true` branch while $not(x > 0)$ for the `false` branch.

$$\begin{array}{l} [\langle \mathcal{H} \mid \begin{array}{l} \mathtt{if}\ x+1 > 0 \\ \mathtt{then}\ 5\ \mathtt{else}\ \mathtt{BAD} \end{array} \mid \begin{array}{l} (\mathtt{if}\ x > 0\ \mathtt{then}\ \bullet) \\ :: (\lambda x.\bullet) :: [\,] \end{array} \mid \begin{array}{l} \forall x : \mathtt{int}, \\ x > 0 \end{array} \rangle; \\ \langle \mathcal{H} \mid \mathtt{UNR} \mid (\mathtt{if}\ x > 0\ \mathtt{else}\ \bullet) :: \mathcal{S} \mid \forall x : \mathtt{int}, not(x > 0) \rangle] \end{array}$$

In the `true` branch, after a few steps, we rebuild the scrutinee $x + 1 > 0$. In this case, $\forall x : \mathtt{int}, x > 0 \Rightarrow x + 1 > 0$ is valid. By [R-s-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:

$$\begin{array}{l} [\langle\!\langle \mathcal{H} \mid 5 \mid \begin{array}{l} (\mathtt{if}\ x > 0\ \mathtt{then}\ \bullet) \\ :: (\lambda x.\bullet) :: [\,] \end{array} \mid \begin{array}{l} \forall x : \mathtt{int}, x > 0, \\ (x + 1 > 0) \end{array} \rangle\!\rangle; \\ \langle\!\langle \mathcal{H} \mid \mathtt{UNR} \mid \begin{array}{l} (\mathtt{if}\ x > 0\ \mathtt{else}\ \bullet) \\ :: (\lambda x.\bullet) :: [\,] \end{array} \mid \begin{array}{l} \forall x : \mathtt{int}, \\ not(x > 0) \end{array} \rangle\!\rangle] \end{array}$$

By [R-match], we combine both simplified branches to rebuild the match-expression:

$$\langle\!\langle \mathcal{H} \mid \mathtt{if}\ x > 0\ \mathtt{then}\ 5\ \mathtt{else}\ \mathtt{UNR} \mid (\lambda x.\bullet) :: [\,] \mid \forall x : \mathtt{int} \rangle\!\rangle$$

We continue to rebuild the expression by [R-lam]:

$$\langle\!\langle \mathcal{H} \mid \lambda x.\ \texttt{if}\ x > 0\ \texttt{then}\ 5\ \texttt{else}\ \texttt{UNR} \mid [\,] \mid \forall x : \texttt{int} \rangle\!\rangle$$

and terminate (by [R-done]) with a syntactically safe expression:

$$\lambda x.\ \texttt{if}\ x > 0\ \texttt{then}\ 5\ \texttt{else}\ \texttt{UNR}.$$

Besides [R-s-save], another transition that saves ctx-info to $\mathcal{L}$ is [R-let-save]. Consider an example:

$$\lambda v.\ \texttt{let}\ y = v + 1\ \texttt{in}\ \texttt{if}\ y > v\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}$$

After a few simplification steps, we have:

$$\langle\!\langle \mathcal{H} \mid v + 1 \mid \begin{array}{l}(\texttt{let}\ y = \bullet\ \texttt{in}\ \texttt{if}\ y > v \\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}) :: (\lambda v.\bullet) :: [\,]\end{array} \mid \forall v : \texttt{int} \rangle\!\rangle$$

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.

$$\langle \mathcal{H} \mid \begin{array}{l}\texttt{if}\ y > v \\ \texttt{then}\ y \\ \texttt{else}\ \texttt{BAD}\end{array} \mid \begin{array}{l}(\texttt{let}\ y = v + 1\ \texttt{in}\ \bullet) \\ :: (\lambda x.\bullet) :: [\,]\end{array} \mid \begin{array}{l}\forall v : \texttt{int}, \\ \forall y : \texttt{int}, \\ y = v + 1\end{array} \rangle$$

Since $\forall v : \texttt{int}, \forall y : \texttt{int}, y = v + 1 \Rightarrow y > v$ is valid, by [R-s-match], we only need to simplify the `true` branch:

$$\langle \mathcal{H} \mid y \mid \begin{array}{l}(\texttt{let}\ y = v + 1\ \texttt{in}\ \bullet) \\ :: (\lambda v.\bullet) :: [\,]\end{array} \mid \begin{array}{l}\forall v : \texttt{int}, \forall y : \texttt{int}, \\ y = v + 1, y > v\end{array} \rangle$$

which leads to the final result $\lambda v.\ \texttt{let}\ y = v + 1\ \texttt{in}\ y$, which is syntactically safe.

5.2 Logicization

$$\begin{array}{rcll}
x, s, i, f & \in & & \textbf{Identifier} \\
\mathit{file} & ::= & \mathit{decl}_1, \ldots, \mathit{decl}_n & \\
\mathit{bty} & ::= & \texttt{int} \mid \texttt{bool} \mid i \mid \texttt{'}i \mid \overrightarrow{\mathit{bty}}\ i & \textbf{Base type} \\
\mathit{lty} & ::= & \mathit{bty} \mid \vec{\mathit{ty}}\ \texttt{->}\ \mathit{bty} & \textbf{Logic type} \\
\mathit{ty} & ::= & \alpha \mid (\mathit{ty}_1, \ldots, \mathit{ty}_n)\ s & \textbf{Types} \\
\mathit{decl} & ::= & \texttt{type}\ \vec{\texttt{'}i}\ s & \\
& \mid & \texttt{logic}\ \vec{i}\ : \mathit{lty} \mid \texttt{axiom}\ i : \phi \mid \texttt{goal}\ i : \phi & \\
\oplus & ::= & + \mid - \mid * \mid / & \\
\odot_t & ::= & = \mid < \mid \leq \mid > \mid \geq & \\
\odot_p & ::= & \texttt{->} \mid \texttt{<->} \mid \texttt{or} \mid \texttt{and} & \\
m & ::= & n \mid x \mid m_1 \oplus m_2 \mid \texttt{-}\ m \mid f\ \vec{m} & \textbf{Term} \\
\phi & ::= & \mathit{true} \mid \mathit{false} \mid f\ \vec{m} & \textbf{Predicate} \\
& \mid & m_1 \odot_t m_2 \mid \phi_1 \odot_p \phi_2 \mid \mathit{not}(\phi) & \\
& \mid & \texttt{forall}\ \vec{x} : \mathit{ty}.\phi \mid \texttt{exists}\ \vec{x} : \mathit{ty}.\phi &
\end{array}$$

Figure 7: Syntax of logic declaration

We now explain the mysterious conversion $[\![.]\!]_f$, which we call *logicization*. Figure 7 gives the abstract syntax of the logical formula supported by an SMT solver named Alt-ergo [8], 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.

```
type 'a list = Nil | Cons of 'a * ('a list)
```

to be converted to Alt-ergo code with `type` and `logic` declarations:

```
type 'a list
logic nil  : 'a list
logic cons : 'a , 'a list -> 'a list
```

As Alt-ergo supports only first order logic (FOL), arguments of a logical function are a tuple, e.g. `'a , 'a list`. The type variable `'a` is assumed universally quantified at top-level. The conversion algorithm for an arbitrary user-defined data type is in Figure 8.

Moreover, we introduce a first order function type:

Data type in language M:

$\texttt{type}\ \vec{\texttt{'}a}\ s = K_1\ \texttt{of}\ \vec{t_1} \mid \cdots \mid K_n\ \texttt{of}\ \vec{t_n}$

Corresponding alt-ergo code: $\texttt{type}\ \vec{\texttt{'}a}\ s$
$\texttt{logic}\ K_1 : \vec{t_1}\ \texttt{->}\ \vec{\texttt{'}a}\ s$
$\vdots$
$\texttt{logic}\ K_n : \vec{t_n}\ \texttt{->}\ \vec{\texttt{'}a}\ s$

Figure 8: Converting data type to Alt-ergo code

```
type ('a, 'b) arrow
```

which allows us to encode the function type in the language M to Alt-ergo's first order type where the `'a` and `'b` refer to a function's input type and output type respectively. We also introduce a logical function `apply`:

```
logic apply : ('a, 'b) arrow , 'a -> 'b
```

where encoding with `apply` is conventional [23]. Converting types in the language M is straight forward (Figure 9).

$$\begin{array}{rcl}
[\![\tau_1 \ldots \tau_n\ T]\!] & = & [\![\tau_1]\!] \ldots [\![\tau_n]\!]\ T \\
[\![\tau_1 \to \tau_2]\!] & = & ([\![\tau_1]\!], [\![\tau_2]\!])\ \texttt{arrow}
\end{array}$$

Figure 9: Converting higher order type to first order type

We now give an example to show what logicization can do.

```
(* val len : 'a list -> int *)
contract len = {x | true} -> {y | y >= 0}
let len s = match s with | []   -> 0
                         | x::u -> 1 + len u

(* val append : 'a list -> 'a list -> 'a list *)
contract append = {xs | true} -> {ys | true}
              -> {len rs = len xs + len ys}
let append xs ys = match xs with
  | []   -> ys
  | x::u -> x :: append u ys
```

The function `len` computes the length of a list and the function `append` appends two lists. Let $e_{\texttt{a}}$ and $t_{\texttt{a}}$ stand for the definition and contract of `append` respectively. Applying only simplification rules (including reduction rules) to $e_{\texttt{a}} \triangleright t_{\texttt{a}}$, we get (R3):

$$\begin{array}{l}
\lambda v_1.\lambda v_2.\texttt{match}\ v_1\ \texttt{with} \\
\mid [\,] \to \texttt{if}\ \texttt{len}\ v_2 = \texttt{len}\ v_1 + \texttt{len}\ v_2\ \texttt{then}\ v_2\ \texttt{else}\ \texttt{BAD}^{l1} \\
\mid x :: u \to\ \ \texttt{if}\ (\texttt{len}\ (x :: \\
\qquad\qquad (\texttt{if}\ \texttt{len}\ (\texttt{append}\ u\ v_2) = \texttt{len}\ u + \texttt{len}\ v_2 \\
\qquad\qquad\ \texttt{then}\ \texttt{append}\ u\ v_2\ \texttt{else}\ \texttt{UNR})) \\
\qquad\qquad = \texttt{len}\ v_1 + \texttt{len}\ v_2) \\
\qquad\quad \texttt{then}\ x :: \texttt{append}\ u\ v_2\ \texttt{else}\ \texttt{BAD}^{l2}
\end{array}$$

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 chanllenge 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 $f = e$, they first η-expand e to get $f = \lambda x_1 \ldots x_n.e'$ where e' does not contain λ; if it is a recursive function, they assume e 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 $\forall \overrightarrow{x}, f(x_1, \ldots, x_n) = [\![e']\!]$ where $[\![.]\!]$ converts an expression to logical form. (On the other hand, [6] uses λ-lifting method: λ-abstractions are translated from inside out, each λ-abstraction is replaced by a call to a newly defined functions. That is to form $\forall \overrightarrow{x}, f_n(x_1, \ldots, x_n) = [\![e']\!]; \ldots; \forall x_1, f = f_1(x_1)$.) This is fine for converting total terms, e.g. $[\![5]\!] = 5$ and $[\![x]\!] = x$, etc., but what are $[\![\texttt{BAD}]\!]$ and $[\![\texttt{UNR}]\!]$? Our key idea is not to convert an expression directly to a corresponding logical term, but form equality with $[\![.]\!]_f$ recursively (defined in Figure 10). The subscript f in $[\![e]\!]_f$ denotes the expression e. Moreover, we perform neither η-expansion (which does not preserve semantics in the presence of non-total terms) nor λ-lifting, and yet we allow arbitrary forms of recursive functions. We have such flexibility because we convert λ-abstraction and partial application directly with the help of `apply`. (Note that our logicization $[\![.]\!]_f$ can also produce HOL formula for interactive proving by replacing $(\texttt{apply}(f, x))$ 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 `BAD`s in R3 are unreachable.

```
logic len: ('a list, int) arrow
logic append: ('a list,
               ('a list,'a list) arrow) arrow

axiom len_def_1 : forall s:'a list. s = nil ->
 apply(len,s) = 0
axiom len_def_2 : forall s:'a list. forall x:'a.
 forall l:'a list. s = cons(x,l) ->
 apply(len,s) = 1 + apply(len,l)

goal app_1 : forall v1,v2:'a list. v1 = nil ->
 apply(len,v2) = apply(len,v1) + apply(len,v2)

goal app_2 : forall v1,v2,l:'a list.forall x:'a.
 v1 = cons(x,l) ->
 apply(len,apply(apply(append,l),v2))
       = apply(len,l) + apply(len,v2) ->
 (exists y:'a list. y = apply(apply(append,l),v2)
   and apply(len,cons(x, y))
         = apply(len,v1) + apply(len,v2))
```

To make an SMT solver's life easier (i.e. multiple small axioms are better than one big axiom), we have two axioms for `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 `->` (in axioms and goals) is a logical implication.

For example, the axiom `len_def_1`, is generated by:

$$
\begin{array}{rl}
 & [\![\lambda s^{\texttt{'a list}}.\, \texttt{match}\ s\ \texttt{with}\ \{\texttt{Nil} \to 0\}]\!]_{\texttt{len}} \\
= & \forall s : \texttt{'a list}. [\![\texttt{match}\ s\ \texttt{with}\ \{\texttt{Nil} \to 0\}]\!]_{(\texttt{apply}(\texttt{len}, s))} \\
= & \forall s : \texttt{'a list}.\ \exists x_0 : \texttt{'a list}. [\![s]\!]_{x_0} \wedge \\
 & \qquad (x_0 = \texttt{nil} \texttt{ -> } [\![0]\!]_{(\texttt{apply}(\texttt{len}, s))}) \\
= & \forall s : \texttt{'a list}.\ \exists x_0 : \texttt{'a list}.\ x_0 = s \wedge \\
 & \qquad (x_0 = \texttt{nil} \texttt{ -> } \texttt{apply}(\texttt{len}, s) = 0)
\end{array}
$$

Let x_0 be s, we get a more readable version (axiom `len_def_1`). An algorithm that partially eliminates redundant existentially quantified variables can be found in [40].

Theorem 6 (Logicization for axioms)**.** *Given definition* $f = e^\tau$, $\exists f : \tau, [\![e]\!]_f$ *is valid.*

Next, what query (i.e. goal) shall we make? All we want is to check if the branch leading to `BAD` 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 `v1=nil` is from the pattern matching `match v1 with {[] -> ....}`; the query is `apply(len,v2) = apply(len,v1) + apply(len,v2)`. The goal `app_1` states the ctx-info $\mathcal{L}$ implies the scrutinee. We have $\mathcal{L} = \forall v_1 : \texttt{'a list}, \forall v_2 : \texttt{'a list}, v_1 = \texttt{nil}$ by [S-lam] and [R-s-save]. The scrutinee is $[\![\texttt{len}\ v_2 = \texttt{len}\ v_1 + \texttt{len}\ v_2]\!]_{\texttt{true}}$. That is, we want to check whether $\texttt{len}\ v_2 = \texttt{len}\ v_1 + \texttt{len}\ v_2$ is equivalent to `true`. Alt-ergo says *valid* for both goals. Thus, we know both $\texttt{BAD}^{l1}$ and $\texttt{BAD}^{l2}$ are not reachable.

$$
\begin{array}{rcl}
\oplus \in [+, -, *, /] & & \odot \in [>, <, =] \\
[\![.]\!]_f & : & \textbf{Expression} \to \textbf{Formula} \\
[\![\texttt{let (rec)}\ f = e]\!]_f & = & [\![e]\!]_f \quad \text{top-level defn} \\
[\![\texttt{BAD}^l]\!]_f & = & \begin{cases} \mathit{true} & \text{for axioms} \\ \mathit{false} & \text{for goals} \end{cases} \\
[\![\texttt{UNR}^l]\!]_f & = & \mathit{false} \\
[\![x]\!]_f & = & f = x \\
[\![n]\!]_f & = & f = n \\
[\![e_1^{\tau_1} \oplus e_2^{\tau_2}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ \exists x_2 : [\![\tau_2]\!], \\
 & & [\![e_1]\!]_{x_1} \wedge [\![e_2]\!]_{x_2} \wedge f = x_1 \oplus x_2 \\
[\![e_1^{\tau_1} \odot e_2^{\tau_2}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \\
 & & \exists x_2 : [\![\tau_2]\!],\ [\![e_2]\!]_{x_2} \wedge \\
 & & ((x_1 \odot_t x_2 \wedge f = \texttt{true}) \vee \\
 & & (\mathit{not}(x_1 \odot_t x2) \wedge f = \texttt{false})) \\
[\![\lambda x^\tau . e]\!]_f & = & \forall x : [\![\tau]\!],\ [\![e]\!]_{(\texttt{apply}(f,x))} \\
[\![\texttt{let}\ x^\tau = e_1\ \texttt{in}\ e_2]\!]_f & = & \exists x : [\![\tau]\!],\ [\![e_1]\!]_x \wedge [\![e_2]\!]_f \\
[\![e_1^{\tau_1}\ e_2^{\tau_2}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \\
 & & \exists x_2 : [\![\tau_2]\!],\ [\![e_2]\!]_{x_2} \wedge \\
 & & f = \texttt{apply}(x_1, x_2) \\
[\![K\ e_1^{\tau_1} \ldots e_n^{\tau_n}]\!]_f & = & \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \cdots \wedge \\
 & & \exists x_n : [\![\tau_n]\!],\ [\![e_n]\!]_{x_n} \wedge \\
 & & f = K\ (y_1, \ldots, y_n) \\
[\![\begin{array}{c} \texttt{match}\ e_0^{\tau_0}\ \texttt{with} \\ \overrightarrow{K\ \overrightarrow{x^\tau} \to e} \end{array}]\!]_f & = & \begin{array}{l} \exists x_0 : [\![\tau_0]\!],\ [\![e_0]\!]_{x_0} \wedge \\ (\bigwedge \overrightarrow{\forall \overrightarrow{x : [\![\tau]\!]}, (x_0 = K\ \overrightarrow{x}) \Rightarrow [\![e]\!]_f}) \end{array}
\end{array}
$$

Figure 10: Convert expression to logical formula

Theorem 7 (Validity preservation: logicization for goals)**.** *For all (possibly open) expression* e^τ, $\exists f : \tau$, *if* $\forall fv(e) : \tau, [\![e]\!]_f$ *is valid and* $e \to e'$ *for some* e', *then* $\forall fv(e') : \tau, [\![e']\!]_f$ *is valid.*

There are a few things to note about logicization.

Syntax abbreviation The Alt-ergo syntax

$$\overrightarrow{\texttt{logic}\ x : lty};\quad \overrightarrow{\texttt{axiom}\ a_i : \phi_i};\quad \overrightarrow{\texttt{goal}\ g_j : \phi_j}$$

is semantically the same as $\forall \overrightarrow{x : lty},\ \overrightarrow{\phi_i} \Rightarrow \overrightarrow{\phi_j}$ where $\overrightarrow{\phi}$ means a conjunction of a set of logical formulae.

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

Crashing functions called in contracts In Figure 10, there are two conversions for `BAD`, *true* for axioms and *false* for goals. For example, we may have:

```
contract g = {x | x /= []} -> {y | 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:

```
logic head : ('a list, 'a) arrow
axiom head_def_1 : forall x:'a list. x=[] -> true
```

```
axiom head_def_2 : forall x,l:'a list.forall y:'a.
    x = cons(y,l) -> apply(head, x) = y
```

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 $[\![e]\!]_f$ collects ctx-info *before* a scrutinee of a `match`-expression, thus, $[\![\texttt{BAD}]\!]_f = [\![\texttt{UNR}]\!]_f = \mathit{false}$, which implies everything. For example:

```
fun x -> let y = if x > 0 then x else UNR in
         if y + 1 > 0 then y + 1 else BAD
```

The ctx-info $\mathcal{L}$ before $y + 1 > 0$ is $\forall x\colon \texttt{int}, \forall y\colon \texttt{int}, (x > 0 \Rightarrow y = x) \wedge (not(x > 0) \Rightarrow \mathit{false})$. So $\mathcal{L} \Rightarrow y + 1 > 0$ is $\forall x\colon \texttt{int}, \forall y\colon \texttt{int}, (x > 0 \Rightarrow y = x) \wedge (not(x > 0) \Rightarrow \mathit{false}) \Rightarrow y + 1 > 0$, which is valid. It means, if $not(x > 0)$ 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, $h_1 = \texttt{if}\ (\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c)\ \texttt{then}\ d\ \texttt{else}\ e$, where a, b, c, d, e are expressions. At program point d, the ctx-info is $(a \Rightarrow b) \wedge (not(a) \Rightarrow c)$[1]. Applying [match-match] to h_1, we get: $h_2 = \texttt{if}\ a\ \texttt{then}\ (\texttt{if}\ b\ \texttt{then}\ d\ \texttt{else}\ e)\ \texttt{else}\ (\texttt{if}\ c\ \texttt{then}\ d\ \texttt{else}\ e)$. The d is duplicated and the ctx-info for the first d is $a \wedge b$ while for the second d is $not(a) \wedge c$. 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 c is `BAD` or `UNR`, the SL machine immediately rebuilds the blame with the rest of the stack, and we get: $\texttt{if}\ a\ \texttt{then}\ (\texttt{if}\ b\ \texttt{then}\ d\ \texttt{else}\ e)\ \texttt{else}\ c$. 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

$$(\lambda v.v + 1)\ (\texttt{if}\ x_1 > 0\ \texttt{then}\ x_1\ \texttt{else}\ \texttt{UNR}^?)$$

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

$$\exists y, (x_1 > 0 \Rightarrow y = x_1 + 1) \wedge (not(x_1 > 0) \Rightarrow \mathit{false}) \qquad [\text{Q2}]$$

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

We have implemented a prototype[2] based on the source code of ocamlc-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 1. Results of preliminary experiments

program	total LOC	Ann LOC	Time (sec)
intro123, neg	23	4	0.08
McCarthy's 91	4	1	0.02
ack, fhnhn	12	2	0.06
arith, sum, max	26	4	0.20
zipunzip	12	2	0.10
OCaml stdlib/list.ml	81	16	0.72

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 $1 \le i \le n$. 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); `sl` is the SL machine; `rmUNR` implements the rule `[rmUNR]` (mentioned earlier in §2).

$$(\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ \texttt{UNR}) \Longrightarrow e_1 \qquad [\texttt{rmUNR}]$$

We have:

$$[\text{SCC}]:\ \texttt{sl}(e_i[(f_j \lhd^{f_j}_{f_i} t_{f_j})/f_j] \rhd^{f_i}_{?} t)$$

$$[\text{DCC}]:\ e_i[(f_j \underset{\texttt{BAD}^{f_i}}{\overset{\texttt{BAD}^{f_j}}{\bowtie}} t_{f_j})/f_j]$$

$$[\text{HCC}]:\ f_i\sharp = \lambda ?.\texttt{rmUNR}(\texttt{sl}(e_i[((f_j\sharp\ \text{“}f_i\text{”}) \lhd^{f_j}_{f_i} t_{f_j})/f_j] \rhd^{f_i}_{?} t))$$

In [HCC], the residual code $f_i\sharp$'s parameter “?” 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 $\texttt{append}\sharp$ is:

$$\begin{array}{l} \lambda ?.\lambda v_1.\lambda v_2.\texttt{match}\ v_1\ \texttt{with} \\ \mid [\,] \to v_2; \\ \mid x :: l \to\ \texttt{if}\ \texttt{len}\ (x :: \texttt{append}\ t\ v_2) = \texttt{len}\ v_1 + \texttt{len}\ v_2 \\ \qquad\qquad \texttt{then}\ x :: \texttt{append}\ t\ v_2\ \texttt{else}\ \texttt{BAD}^l \end{array}$$

which says that we only have to check postcondition for the second branch. (If all `BAD`s 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,* $\quad (e \underset{r_2}{\overset{r_1}{\bowtie}} t) \underset{r_4}{\overset{r_3}{\bowtie}} t = e \underset{r_4}{\overset{r_1}{\bowtie}} t$.

[1] To illustrate the idea with less cluttered form, we omit the conversion notation $[\![.]\!]_f$ for *a*, *b*, *c*, *d*, *e*.

[2] http://gallium.inria.fr/˜naxu/research/hcc.html

Precondition of a function is checked at caller sites. An $f_j\sharp$ is the simplified $f_j \triangleright^{f_j}_{f_i} t_{f_j}$, inspecting [HCC], each f_j at caller sites is replaced by $(f_j \triangleright^{f_j}_{f_i} t_{f_j}) \triangleleft^{f_j}_{f_i} t_{f_j}$, which is $(f_j \underset{\mathtt{UNR}^{f_i}}{\overset{\mathtt{BAD}^{f_j}}{\bowtie}} t_{f_j}) \underset{\mathtt{BAD}^{f_i}}{\overset{\mathtt{UNR}^{f_j}}{\bowtie}} t_{f_j}$. By the telescoping property, we have:

$$(f_j \underset{\mathtt{UNR}^{f_i}}{\overset{\mathtt{BAD}^{f_j}}{\bowtie}} t_{f_j}) \underset{\mathtt{BAD}^{f_i}}{\overset{\mathtt{UNR}^{f_j}}{\bowtie}} t_{f_j} = f_j \underset{\mathtt{BAD}^{f_i}}{\overset{\mathtt{BAD}^{f_j}}{\bowtie}} t_{f_j} \qquad \text{[T1]}$$

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 `[rmUNR]` because all `UNR`s are indeed unreachable as $\mathtt{BAD}^l$ is invoked before $\mathtt{UNR}^l$ for the same l. That is, ($\mathtt{if}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ \mathtt{BAD}^l$) is invoked before ($\mathtt{if}\ p\ \mathtt{then}\ e\ \mathtt{else}\ \mathtt{UNR}^l$) for the same p, maybe different e. So it is safe to apply the rule `[rmUNR]` even if p diverges or crashes because the same p in ($\mathtt{if}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ \mathtt{BAD}$) diverges or crashes first. It is easy to see if $t = \{x \mid p\}$. If $t = t_1 \to t_2$, then $(e \underset{\mathtt{UNR}^{f_i}}{\overset{\mathtt{BAD}^{f_j}}{\bowtie}} t_1 \to t_2) \underset{\mathtt{BAD}^{f_i}}{\overset{\mathtt{UNR}^{f_j}}{\bowtie}} t_1 \to t_2$ expands to $\mathtt{let}\ y = e\ \mathtt{in}$

$$\lambda v_2.((\lambda v_1.(y\ (v_1 \underset{\mathtt{BAD}^{f_j}}{\overset{\mathtt{UNR}^{f_i}}{\bowtie}} t_1)) \underset{\mathtt{UNR}^{f_i}}{\overset{\mathtt{BAD}^{f_j}}{\bowtie}} t_2)\ (v_2 \underset{\mathtt{UNR}^{f_j}}{\overset{\mathtt{BAD}^{f_i}}{\bowtie}} t_1)) \underset{\mathtt{BAD}^{f_i}}{\overset{\mathtt{UNR}^{f_j}}{\bowtie}} t_2$$

Focusing on the `BAD`s and `UNR`s above $\bowtie$, inspecting [P1] in Figure 4, $\mathtt{BAD}^{f_j}$ is invoked before $\mathtt{UNR}^{f_j}$ and $\mathtt{BAD}^{f_i}$ is invoked before $\mathtt{UNR}^{f_i}$. 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 always has its limitation, 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. subcontract 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 $e \triangleright t$ to be crash-free in one-go and do not have to check t to be crash-free separately. In practice, the $\mathcal{H}$ and $\mathcal{L}$ 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 $\{_t \; _t\}$ for type checking and $\{_s \; _s\}$ 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 $\bullet$ 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 Pouillard, Martin Berger, Simon Peyton Jones, Michael Greenberg for their feedback.

References

[1] N. Ayache and J.-C. Filliatre. Combining the Coq proof assistant with first-order decision procedures. Unpublished, 2006. URL `http://www.lri.fr/~filliatr/publis/coq-dp.ps`.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *CASSIS*, LNCS 3362, 2004.

[3] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *ESOP*, pages 18–37, 2011.

[4] A. M. Ben-Amram and C. S. Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29:5:1–5:37, January 2007.

[5] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33:8:1–8:45, February 2011.

[6] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *CADE*, pages 116–130, 2011.

[7] M. Blume and D. A. McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4-5):375–414, 2006.

[8] S. Conchon, E. Contejean, and J. Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006. URL `http://ergo.lri.fr/papers/ergo.ps`.

[9] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *CADE*, pages 263–278, 2007.

[10] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'11, pages 215–226, New York, NY, USA, 2011.

[11] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS'10: International Conference on Formal Verification of Object-Oriented Software*, pages 10–30, 2010.

[12] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Functional and Logic Programming*, pages 226–241. Springer Berlin / Heidelberg, 2006.

[13] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, 2002.

[14] C. Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 245–256, 2006.

[15] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

[17] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/363235.363259. URL `http://doi.acm.org/10.1145/363235.363259`.

[19] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 191–202, 2004.

[20] K. Honda, M. Berger, and N. Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *the 33rd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 360–371, 2006.

[21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods*, pages 41–55, 2011.

[22] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *the 15th international conference on Computer Aided Verification CAV*, pages 262–274, 2011.

[23] M. Kerber. How to prove higher order theorems in first order logic. In *IJCAI*, pages 137–142, 1991.

[24] Y. P. Khoo, B.-Y. E. Chang, and J. S. Foster. Mixing type checking and symbolic execution. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 436–447, New York, NY, USA, 2010. ACM.

[25] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32:6:1–6:34, February 2010. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1667048.1667051. URL `http://doi.acm.org/10.1145/1667048.1667051`.

[26] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 416–428, 2009.

[27] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and cegar for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 222–233, New York, NY, USA, 2011.

[28] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. ACM.

[29] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 312–327, 2010.

[30] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.

[31] M. Might. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 185–198, 2007.

[32] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *ICFP*, pages 62–73. ACM, 2006.

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

[34] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 587–598, 2011.

[35] Y. Régis-Gianas and F. Pottier. A hoare logic for call-by-value functional programs. In P. Audebaud and C. Paulin-Mohring, editors, *MPC*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, 2008.

[36] D. Sereni and N. D. Jones. Termination analysis of higher-order functional programs. In *proceedings of the 3rd Asian Symposium on Program. Lang. and Systems (APLAS)*, pages 281–297, 2005.

[37] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, 2011.

[38] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.

[39] D. N. Xu. Extended static checking for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 48–59, 2006.

[40] D. N. Xu. Hybrid contract checking. INRIA research report, 2011. URL `http://gallium.inria.fr/~naxu/research/hcc-tr.pdf`.

[41] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 41–52, 2009.

[42] N. Xu. *Static Contract Checking for Haskell*. Ph.D. thesis, Aug. 2008.