# Hybrid Contract Checking via Symbolic Simplification

Dana N. Xu

INRIA Paris-Rocquencourt

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 a popular approach to program verification and error discovery. We formalize and implement a hybrid (static and dynamic) contract checker for a subset of OCaml. The key technique is symbolic simplification, which makes integrating static and dynamic contract checking easy and effective. Our technique statically checks contract satisfaction 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*** functional language, verification, debugging

***Keywords*** contract semantics, static, dynamic, hybrid, contract checking, symbolic simplification

## 1. Introduction

Constructing reliable software is difficult. Formulating and checking (statically or dynamically) logical assertions [2, 5, 15, 17, 36], especially in the form of contracts [7, 12, 13, 29, 40], is one popular approach to error discovery. Static contract checking can catch all contract violations but may raise false alarms and can only check restricted properties; dynamic checking can check more expressive properties but consumes run-time cycles and only checks the paths actually executed, and so is not complete. 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 `f1` will return a non-negative number whenever it is applied to a function that returns a non-negative number when given a non-negative number. Both a static checker and a dynamic checker are able to report that `f1` fails its postcondition: a static checker relies on the unsoundness of $\forall g : \texttt{int} \rightarrow \texttt{int},\ (g\ 1) \geq 0 \Rightarrow (g\ 1) - 1 \geq 0$ while a 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 $\forall x : \texttt{int}, x \geq 0 \Rightarrow x - 1 \geq 0$ does not hold. 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. As static and dynamic checking can be complementary, we may want to invoke a dynamic checker when the outcome is (c). This ensures that no contract violations can escape while maintaining expressiveness.

Following the formalization in [40], but this time for a strict language, we first give a denotational semantics for contract satisfaction, i.e., we define what it means for an expression $e$ to satisfy its contract $t$ (written $e \in t$) without knowing how to check it. Next, we define a wrapper $\triangleright$ that takes $e$ and $t$ and produces a term $e \triangleright t$ with contract checks inserted at appropriate places in $e$. If a contract check is violated, a special constructor $\texttt{BAD}^l$ signals the violation where the label $l$ precisely captures the function at fault. All we have to do is to check the reachability of $\texttt{BAD}^l$ in the term $e \triangleright t$. We symbolically simplify the term $e \triangleright t$, aiming to simplify `BAD`s away. If some `BAD` constructors remain, 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 going to dynamic checking. Such unreachable code, however, is essential during static checking. We show the correctness of this pruning (§6) with the telescoping property studied (but not used for such purpose) in [7, 40].
- We define $e \in t$ and $e \triangleright t$ and prove a theorem "$e \triangleright t$ is crash-free $\iff$ $e \in t$" (§4). "Crash-free" means `BAD` is not reachable under any context. Such a formalization is tricky and its correctness proof is non-trivial. We rework the proofs from [41] 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 checking is automatic and *modular* and we prove its soundness. Moreover, the SL machine produces *residual* code for dynamic checking.
- We design a *logicization* technique that transforms expressions to logical formulae. The key contribution is to deal with non-total terms (§5).

## 2. Overview

Assertions [17] 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 provide precise



*PEPM'12,* January 23–24, 2012, Philadelphia, PA, USA.

blaming of the function at fault. Contracts [29] and higher-order contracts [12] were to be checked at run-time when they were first introduced. 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 h1 = inc 0
```

A dynamic checker wraps the `inc` in `h1` with its contract $t_{\texttt{inc}}$ (a shorthand for the contract of `inc`):

$$\texttt{let h1 = }(\texttt{inc} \underset{\texttt{BAD}^{l'}}{\overset{\texttt{BAD}^{l}}{\bowtie}} t_{\texttt{inc}})\, 0$$

where $l$ is $(2, 5, \text{“}\texttt{inc}\text{”})$ the (row,col) source location where `inc` is defined and $l'$ is $(3, 10, \text{“}\texttt{h1}\text{”})$ the source location of the call site with caller's name. This wrapped `h1` expands to:

$$\begin{array}{l}(\lambda x_1.\ \texttt{let}\ y =\ \texttt{inc}\ (\texttt{let}\ x = x_1\ \texttt{in}\\ \qquad\qquad\qquad\quad \texttt{if}\ x > 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{BAD}^{(3,10,\text{“}\texttt{h1}\text{”})})\\ \quad\ \texttt{in}\ \texttt{if}\ y > 0\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}^{(2,5,\text{“}\texttt{inc}\text{”})}\ )\ 0\end{array}$$

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 blame is raised. In this case, `h1` calls `inc` with 0, which fails `inc`'s precondition. Running the above wrapped code, we get $\texttt{BAD}^{(3,10,\text{“}\texttt{h1}\text{”})}$, which blames `h1`.

With the DCC algorithm, 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:

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

, 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.

Later, [7, 40] give formal declarative semantics for contract satisfaction that not only allow us to prove the correctness of DCC against this semantics, but also to check contracts statically.

The essence of *static contract checking* (SCC) is:

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

The $\rhd$ (“ensures”) and the $\lhd$ (“requires”) are dual to each other. The special constructor `UNR` (“unreachable”), does not raise blame, but stops execution. (Those who are 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 is performed at the definition site of each function. For example, $(\lambda v.v + 1) \rhd t_{\texttt{inc}}$ expands to:

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

At the definition site of a function, $f = e$, we assume $f$'s precondition 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 $x$ gives:

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

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). In this section, we present formulae in higher-order logic while §5 gives the details of the generation of formulae in FOL. For the two subexpressions of the RHS of $y$, we have:

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

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

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

We check the validity of a formula collected from the path to $\texttt{BAD}^{(2,5,\text{“}\texttt{inc}\text{”})}$, i.e., $\forall x_1, \text{Q1} \Rightarrow y > 0$, by consulting an SMT solver. Since it is valid, we know that 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 $(\lambda g.(g\ 1) - 1) \rhd t_{\texttt{f1}}$, which expands to:

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

After applying some conventional simplification rules, we have:

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

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 $\forall x_1, \forall z, (\exists y, y = x_1\ 1 \wedge (y \geq 0 \Rightarrow z = y - 1)) \Rightarrow z \geq 0$, 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 $\{x \mid \texttt{true}\}$. Suppose $\texttt{zut} \lhd \{x \mid \texttt{true}\} = \texttt{zut}$, 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:

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

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 so we do not elaborate it 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):

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

leaving one BAD. 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, simplifying a bit, we get:

$$\begin{array}{l}
\texttt{f1}\sharp = \lambda x_1.\ \texttt{let}\ z = (\texttt{let}\ y = (x_1\ 1)\ \texttt{in}\ y - 1)\ \texttt{in} \\
\qquad\qquad\quad \texttt{if}\ z \geq 0\ \texttt{then}\ z\ \texttt{else}\ \texttt{BAD}^{(4,5,\text{“}f_1\text{”})} \\
\texttt{f3}\sharp = \ \texttt{f1}\sharp\ \ (\lambda x_2.\texttt{let}\ y = \texttt{zut}\ x_2\ \texttt{in} \\
\qquad\qquad\quad \texttt{if}\ y \geq 0\ \texttt{then}\ y\ \texttt{else}\ \texttt{BAD}^{(7,10,\text{“}f_3\text{”})})
\end{array}$$

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 that $e$ 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, while if there is a BAD in $e'$, we give error (or warning) message for a definite (or potential) bug at compile-time. (4) For a function $f$ not satisfying its contract, 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, and is a subset of OCaml with parametric polymorphism.

## 3.1 Syntax

Figure 1 gives the syntax of language M. A program contains a set of data type declarations, contract declarations and function definitions. Expressions include integers $n$, variables, lambda abstractions, applications, constructors and match expressions. 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 [21] to infer its contract. Even if [21] is not modular, it is enough to infer a contract for a local function.) Pairs are a special case of constructed terms. A local let expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is syntactic sugar for $(\lambda x.e_2)\ e_1$. An if expression $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ is syntactic sugar for $\texttt{match}\ e_0\ \texttt{with}\ \{\texttt{true} \to e_1; \texttt{false} \to e_2\}$.

We assume all top-level functions are given a contract. Contract checking is done after the type checking phase in a compiler so we

$x, f \in$ **Variables** $\quad T \in$ **Type constructors**
$\qquad\qquad\qquad\qquad K \in$ **Data constructors**

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

**Figure 1.** Syntax of the language M

assume all expressions, contexts, and contracts are well-typed and use the type information (presented as a superscript, e.g., $e^\tau$ or $t^\tau$) whenever necessary. Type-checking material is in [39].

The two contract exceptions (also called blames) $\texttt{BAD}^l$ and $\texttt{UNR}^l$ are adapted from [40]. They are for internal usage, and are not visible to programmers. The label $l$ captures source location and function name, which are useful for error reporting as well as for the 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 Error
              | x::l -> x
```

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

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, $x\colon \{y \mid y > 0\} \to \{z \mid z > x\}$ says that, the input is greater than 0 and the output is greater than the input. We can use a shorthand $\{x \mid x > 0\} \to \{z \mid z > x\}$ by assuming $x$ scopes over the RHS of $\to$. The $\to$ is right associative. Similarly, dependent tuple contracts allow us to describe dependency between two components of a tuple. For example, $(x\colon \{y \mid y > 0\}, \{z \mid z > x\})$ 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., $k\colon (\{x \mid x > 0\} \to \{y \mid y > x\}) \to \{z \mid k\ 5 > z\}$ for a function `let f g = g 2`.

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

$$\frac{e_1 \to e_2}{\mathcal{C}[\![e_1]\!] \to \mathcal{C}[\![e_2]\!]}\ \text{[E-ctx]} \qquad \frac{\texttt{let (rec)}\ f = e \in \Delta}{f \to e}\ \text{[E-top]}$$

$$\mathcal{C}[\![r]\!] \to r \quad \text{[E-exn]} \qquad (\lambda x.e)\ val \to e[val/x] \quad \text{[E-beta]}$$

$$\texttt{match}\ K\ \overrightarrow{val}\ \texttt{with}\ \{..,K\ \overrightarrow{x} \to e;..\} \to e[\overrightarrow{val/x}] \quad \text{[E-match]}$$

**Figure 2.** Semantics of the language M

### 3.2 Operational semantics

The semantics of our language is given by the reduction rules in Figure 2. For a top-level function, we fetch its definition from the evaluation environment $\Delta$, which maps a variable to its definition. We adapt some basic definitions from [40]. 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 $r$. The notation $(\mathcal{C}[\![e]\!])^{\texttt{bool}}$ means $\mathcal{C}[\![e]\!]$ is closed and well-typed.

**Definition 1** (Semantically Equivalent)**.** *Two expressions $e_1$ and $e_2$ are semantically equivalent, namely $e_1 \equiv_s e_2$, iff $\forall \mathcal{C}$, $(\mathcal{C}[\![e_i]\!])^{\texttt{bool}}$ for $i = 1, 2$, $r \in \{\texttt{BAD}, \texttt{UNR}\}$, $\quad \mathcal{C}[\![e_1]\!] \to^* r \iff \mathcal{C}[\![e_2]\!] \to^* r$*

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 $e \to^*$ 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 $e \to^*$ UNR, or there is no value $val$ such that $e \to^* val$.*

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 $\lambda x.$ (BAD$, x$) is not.

**Definition 5** (Crash-free Expression)**.** *A (possibly open) expression $e$ is crash-free iff : $\forall \mathcal{C}$,* BAD $\notin_s \mathcal{C}$ *and $(\mathcal{C}[\![e]\!])^{\texttt{bool}} \Rightarrow \mathcal{C}[\![e]\!] \not\to^*$* BAD

The quantified context $\mathcal{C}$ serves the usual role of a probe that tries to provoke $e$ into crashing. A crash-free expression may not be syntactically safe, e.g., $\lambda x.$if $x * x \geq 0$ then $x + 1$ else 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 $\texttt{BAD}^l$ and $\texttt{UNR}^l$ are closed expressions even if $l$ is not explicitly bound.

## 4. Contracts

Inspired by [40], we design a contract satisfaction and checking algorithm for a strict language. As diverging contracts make dynamic contract checking unsound (explained in §4.2) and we do hybrid checking, we focus on total contracts.

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

**Figure 3.** Contract Satisfaction ($e \in t$)

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

*$\quad$ $t$ is $\{x \mid p\}$ and $\lambda x.p$ is **tl** (i.e., crash-free, terminating)*
*or $\;$ $t$ is $x\colon t_1 \to t_2$ and $t_1$ is **tl** and $\forall val \in t_1, t_2[val/x]$ is **tl***
*or $\;$ $t$ is $(x\colon t_1, t_2)$ and $t_1$ is **tl** and $\forall val \in t_1, t_2[val/x]$ is **tl***
*or $\;$ $t$ is* Any

Our definition of total contract is different from that in [7], but close to the crash-free contract in [40] with an additional condition that $\lambda x.p$ is a terminating function. For example, contract $\{x \mid x \neq$ [] $\} \to \{y \mid \texttt{head}\ x > y\}$ is total in our framework because head $x$ does not crash for all $x$ satisfying $\{x \mid x \neq$ []$\}$. 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 $t$" ($e \in t$) in Figure 3. Here are some consequences: (1) a divergent expression satisfies any contract, hence all contracts are inhabited; (2) only crash-free expressions satisfy a predicate contract; (3) any expression satisfies contract Any; (4) BAD only satisfies contract Any.

One difference from [40] is that, we do not allow $p[e/x]$ in [A1] to diverge while [40] allows because they only do static checking. We support dependent tuple contracts, that are not in [7, 40]. One difference from [7] is that, they say that a crashing expression does not satisfy any contract; we say that a crashing expression satisfies the universal contract Any. Having a top ordering contract Any is debated in [11]. We define a subcontract ordering as follows.

**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 $\forall e,\ e \in t_1 \Rightarrow e \in t_2$*

For example, we have $\{x \mid$ true$\} \leqq$ Any, but not vice versa. The Any is like (but not the same as) $\forall \alpha, \alpha$. Consider:

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

In [7] and other refinement type checking framework [5, 24, 36], they give function like fail a function contract $\{x \mid$ false$\} \to \{x \mid$ true$\}$ so that the precondition $\{x \mid$ false$\}$ allows their system to blame all the callers of fail. Using a function contract for a non-function type is somewhat ad hoc. More discussion on the contract Any can be found in [39].

### 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, whose full versions are $\triangleright^{l_1}_{l_2}$ and $\triangleleft^{l_1}_{l_2}$ respectively. The wrapped expression $e \overset{r_1}{\underset{r_2}{\bowtie}} t$ (defined in Figure 4) 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 raises $r_2$ if the wrapped term is used in a way that violates $t$.

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

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

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

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x\!:\!t_1 \to t_2 = \texttt{let}\ y = e\ \texttt{in} \qquad \text{[P2]}$$
$$\lambda x_1.((y\ (x_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_2}{\underset{\underline{r_1}}{\bowtie}} t_1)/x])$$

$$e \overset{r_1}{\underset{r_2}{\bowtie}} (x\!:\!t_1, t_2) = \texttt{match}\ e\ \texttt{with} \qquad \text{[P3]}$$
$$(x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1, x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{\underline{r_2}}{\underset{\underline{r_1}}{\bowtie}} t_1)/x])$$

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

**Figure 4.** Contract checking with the wrappers

**Theorem 1** (Soundness of contract checking)**.** *For all closed expressions $e^\tau$ and closed, terminating contracts $t^\tau$,*

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

The superscript $\tau$ says both $e$ and $t$ are well-typed and have the same type $\tau$. Note that if $t$ is terminating and $e \triangleright t$ is crash-free, then $t$ is total. See [39] for a full proof and a completeness theorem. Basically, we rework the proofs in [41] for a strict language.

Unlike [12], which assumes there are no exceptions in contracts, our checking algorithm detects contract exceptions in *contracts*. The term $t_2[(x_1 \overset{r_2}{\underset{\underline{r_1}}{\bowtie}} t_1/x]$ in [P2] and [P3] says that, each (function) call *in a contract* is wrapped 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}`. The $\underline{r_i}$ says that the label of $r_i$ is updated: $\underline{r_1}$'s label is the call site of $x_1$ in $t_2$ and the name of the contract; $\underline{r_2}$'s label is the location of $x_1$ : and the name of $k$. Correctness of blaming (i.e., the correct labels for $r_i$) is not the focus of this paper. However, our proof [39] is different from that in [7] and the proof in [7] works because they use an ad hoc fix, i.e., using `UNR` instead of $\underline{r_1}$.

***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 []
```

$\texttt{fb} \triangleright t_{\texttt{fb}}$ is $\lambda x_1.((\lambda x.\texttt{head}\ [\,])\ (\texttt{if}\ \texttt{loop}\ x_1\ \texttt{then}\ x_1\ \texttt{else}\ \texttt{BAD}))$, 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, 27, 35] to build an efficient automatic termination checker.

## 5. Static contract checking and residualization

The Theorem 1 in §4.2 says that, to check contract satisfaction, we can check the reachability of `BAD` in $e \triangleright t$ as each `BAD` signals a contract violation. We introduce an SL machine (Figure 5) which tries to simplify away the `BAD`s in an expression. The novelty of our work is to combine symbolic *simplification* and contextual information (ctx-info) synthesis with *logical* store in order 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 $\langle \mathcal{H} \,|\, e \,|\, \mathcal{S} \,|\, \mathcal{L}\rangle$ is to *simplify* $e$ and a 4-tuple $\langle\!\langle \mathcal{H} \,|\, e \,|\, \mathcal{S} \,|\, \mathcal{L}\rangle\!\rangle$ is to *rebuild* $e$ 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 $\phi$ is a predicate in Figure 6.

*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.*

### 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 for the CEK machine [14]), 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 $\mathcal{S}$ is

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

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-m-match], [S-match-let] implement the conventional simplification rules. Here, $\overrightarrow{x}$ abbreviates a sequence of $x_1, \ldots, x_n$. We use `let` instead of lambda for easy reading. Rules [S-letL] and [S-matchL] push the argument into the let-body and match-body respectively; rules [S-letR] and [S-matchR] push the function into the let-body and match-body. The rules [S-m-match] and [S-match-let] are to make an expression less nested. Rule [S-K-match] allows us to simplify an expression like $\texttt{match Some}\ e\ \texttt{with}\ \{\texttt{Some}\ x \to 5; \texttt{None} \to \texttt{BAD}\}$ to $\texttt{let}\ x = e\ \texttt{in}\ 5$ which is crash-free.

What does *rebuild* do? It unwinds the stack. If the stack is empty ([R-done]), indicating the end of the whole simplification process, we return the expression. Otherwise, we examine the stack frame. 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 the next 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.

$$
\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 : \llbracket \tau \rrbracket \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 \mathtt{match}\ e_0\ \mathtt{with}\ alts \mid \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_0 \mid (\mathtt{match}\ \bullet\ \mathtt{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 \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid (\bullet\ e) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \mathtt{let}\ x = e_1\ \mathtt{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}(\mathtt{match}\ e_0\ \mathtt{with} \\ \overrightarrow{K\ \overrightarrow{x} \to e_i})\end{array} \mid (\bullet\ e) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \mathtt{match}\ e_0\ \mathtt{with}\ \overrightarrow{K\ \overrightarrow{x} \to 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\ (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \mathtt{let}\ x = e_1\ \mathtt{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\ (\mathtt{match}\ e_0\ \mathtt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \mathtt{match}\ e_0\ \mathtt{with}\ \overrightarrow{K\ \overrightarrow{x} \to 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}\mathtt{match}\ e_0\ \mathtt{with} \\ \overrightarrow{K\ \overrightarrow{x} \to e}\end{array} \mid (\mathtt{match}\ \bullet\ \mathtt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \begin{array}{l}\mathtt{match}\ e_0\ \mathtt{with} \\ \overrightarrow{K\ \overrightarrow{x} \to \mathtt{match}\ e\ \mathtt{with}\ alts}\end{array} \mid \mathcal{S} \mid \mathcal{L} \rangle & \text{[S-m-match]} \\
\text{if } x \notin fv(alts), & & & \\
\langle \mathcal{H} \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid (\mathtt{match}\ \bullet\ \mathtt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle & \rightsquigarrow & \langle \mathcal{H} \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ \mathtt{match}\ e_2\ \mathtt{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 \mathtt{match}\ e\ \mathtt{with}\ K\ \overrightarrow{x} \to (\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]} \\
\text{Rules below: } a \notin \{\mathtt{BAD}^l, \mathtt{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 (\mathtt{match}\ \bullet\ \mathtt{with}\ \{\ldots; K\ \overrightarrow{x} \to e; \ldots\}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid \mathtt{let}\ \overrightarrow{x = a}\ \mathtt{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 : \llbracket \tau \rrbracket}, \llbracket a \rrbracket_{(K\ \overrightarrow{x})}), & & & \\
\langle\!\langle \mathcal{H} \mid a \mid (\mathtt{match}\ \bullet\ \mathtt{with}\ \overrightarrow{K\ \overrightarrow{x^{\tau}} \to e}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L}, \forall \overrightarrow{x : \llbracket \tau \rrbracket}, \llbracket a \rrbracket_{(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 : \llbracket \tau \rrbracket}, \llbracket a \rrbracket_{(K\ \overrightarrow{x})}), & & & \\
\langle\!\langle \mathcal{H} \mid a \mid (\mathtt{match}\ \bullet\ \mathtt{with}\ \overrightarrow{K\ \overrightarrow{x^{\tau}} \to e}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \overrightarrow{\langle \mathcal{H} \mid e \mid \begin{array}{l}(\mathtt{match}\ a\ \mathtt{with}\ K\ \overrightarrow{x^{\tau}} \\ \to (\bullet, \mathcal{S}, \mathcal{L})) :: [\,]\end{array} \mid \begin{array}{l}\mathcal{L}, \forall \overrightarrow{x : \llbracket \tau \rrbracket}, \\ \llbracket a \rrbracket_{(K\ \overrightarrow{x})}\end{array} \rangle} & \text{[R-s-save]} \\
\overrightarrow{\langle\!\langle \mathcal{H} \mid a \mid (\mathtt{match}\ a_0\ \mathtt{with}\ K\ \overrightarrow{x} \to (\bullet, \mathcal{S}, \mathcal{L})) :: \mathcal{S}' \mid \mathcal{L}' \rangle\!\rangle} & \rightsquigarrow & \langle\!\langle \mathcal{H} \mid \mathtt{match}\ a_0\ \mathtt{with}\ \overrightarrow{K\ \overrightarrow{x} \to 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 (\mathtt{let}\ x^{\tau} = \bullet\ \mathtt{in}\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle & \rightsquigarrow & \langle \mathcal{H} \mid e_2 \mid (\mathtt{let}\ x = a\ \mathtt{in}\ \bullet) :: \mathcal{S} \mid \mathcal{L}, \forall x \colon \llbracket \tau \rrbracket, \llbracket a \rrbracket_x \rangle & \text{[R-let-save]}
\end{array}
$$

**Figure 5.** SL machine

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:

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

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

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

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

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

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 $\exists (K\ \overrightarrow{x}),\ \mathcal{L} \Rightarrow \llbracket a \rrbracket_{(K\ \overrightarrow{x})}$" checks if there is any branch $K\ \overrightarrow{x}$ 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 $\llbracket a \rrbracket_{(K\ \overrightarrow{x})}$ (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.

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

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:

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

$$
\begin{array}{rcll}
x, s, i & \in & \textbf{Identifier} & \\
file & ::= & decl_1, \ldots, decl_n & \\
ty & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{'}\alpha \mid \overrightarrow{ty}\ s & \textbf{Types} \\
lty & ::= & ty \mid \overrightarrow{ty}\ \texttt{->}\ ty & \textbf{Logic type} \\
decl & ::= & \texttt{type}\ \overrightarrow{\texttt{'}\alpha}\ s & \\
 & \mid & \texttt{logic}\ \vec{i}\ : lty \mid \texttt{axiom}\ i : \phi \mid \texttt{goal}\ i : \phi & \\
\oplus & ::= & + \mid - \mid * \mid / & \\
\odot_t & ::= & = \mid < \mid \leq \mid > \mid \geq & \\
\odot_p & ::= & \texttt{->} \mid \texttt{<->} \mid \texttt{or} \mid \texttt{and} & \\
m & ::= & n \mid x \mid m_1 \oplus m_2 \mid \texttt{-}\ m \mid x(\overrightarrow{m}) & \textbf{Term} \\
\phi & ::= & true \mid false \mid f\ \overrightarrow{m} & \textbf{Predicate} \\
 & \mid & m_1 \odot_t m_2 \mid \phi_1 \odot_p \phi_2 \mid not(\phi) & \\
 & \mid & \texttt{forall}\ \vec{x} : ty.\phi \mid \texttt{exists}\ \vec{x} : ty.\phi &
\end{array}
$$

**Figure 6.** Syntax of logic declaration

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

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

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

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

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]. We refer readers to [39] for more examples.

**Theorem 2** (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 [14], 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 3** (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 3 (in [39]) 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 4** (Soundness of static contract checking)**.** *For all closed expression $e$, and closed and terminating contract $t$,*

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

*Proof.* By Theorem 3, Lemma 1 and Theorem 1. □

For open expressions and open contracts, see [39].

### 5.2 Logicization

We now explain the conversion $[\![.]\!]_f$, which we call *logicization*. Figure 6 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 us to represent data type declaration, e.g.,

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

in Alt-ergo code with `type` and `logic` declarations:

Data type declaration in language M:

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

Its alt-ergo code: $\texttt{type}\ \overrightarrow{\texttt{'}a}\ s$

$$\overrightarrow{\texttt{logic}\ K : \overrightarrow{t}\ \texttt{->}\ \overrightarrow{\texttt{'}a}\ s}$$

**Figure 7.** Converting data type to Alt-ergo code

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

**Figure 8.** Converting OCaml types to logic type

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

As Alt-ergo supports only first-order logic (FOL), the arguments of a logical function (e.g., `cons`) are given as a tuple. The type variable `'a` is assumed universally quantified at top-level. The conversion algorithm for an arbitrary user-defined data type is in Figure 7.

A conventional way [22] to encode higher-order function to FOL is to define a type `arrow` and a logical function `apply`:

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

where the `'a` and `'b` refer to a function's input and output type respectively. Converting types in the language M is easy (Figure 8). Base types `int` and `bool` are data types with no parameter.

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):

$$
\begin{array}{l}
\lambda v_1.\lambda v_2.\texttt{match}\ v_1\ \texttt{with} \\
\mid [\,] \to \texttt{if}\ \texttt{len}\ v_2 = \texttt{len}\ v_1 + \texttt{len}\ v_2\ \texttt{then}\ v_2\ \texttt{else}\ \texttt{BAD}^{l1} \\
\mid x :: u \to\ \ \texttt{if}\ (\texttt{len}\ (x :: \\
\qquad\qquad (\texttt{if}\ \texttt{len}\ (\texttt{append}\ u\ v_2) = \texttt{len}\ u + \texttt{len}\ v_2 \\
\qquad\qquad\ \texttt{then}\ \texttt{append}\ u\ v_2\ \texttt{else}\ \texttt{UNR})) \\
\qquad\quad = \texttt{len}\ v_1 + \texttt{len}\ v_2) \\
\qquad \texttt{then}\ x :: \texttt{append}\ u\ v_2\ \texttt{else}\ \texttt{BAD}^{l_2}
\end{array}
$$

The simplification approach in [38] and the model-checking approach in [33] involve inlining top-level functions, while we do not. Instead, we axiomatize the top-level function definitions that were called in contracts and lift expressions under checking to logic level and consult an SMT solver. The challenge is to deal with non-total expressions (e.g., `BAD`) in our source code. In the literature about converting functional code (in an interactive theorem prover) to SMT formulae [1, 6, 9, 28], expressions are converted to a logical form directly. In [1], given a non-recursive function definition $f = e$, they first $\eta$-expand $e$ to get $f = \lambda x_1 \ldots x_n.e'$ where $e'$ does not contain $\lambda$; 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 performs an immediate case-analysis over one of its arguments. Then, they form $\forall \overrightarrow{x}, f(x_1, \dots, x_n) = \llbracket e' \rrbracket$ where $\llbracket . \rrbracket$ converts an expression to logical form. (On the other hand, [6] uses $\lambda$-lifting method: $\lambda$-abstractions are translated from inside out, where each $\lambda$-abstraction is replaced by a call to a newly defined functions, so $\forall \overrightarrow{x}, f_n(x_1, \dots, x_n) = \llbracket e' \rrbracket; \dots; \forall x_1, f = f_1(x_1)$ .) This is fine for converting total terms, e.g., $\llbracket 5 \rrbracket = 5$ and $\llbracket x \rrbracket = x$, etc., but what are $\llbracket \texttt{BAD} \rrbracket$ and $\llbracket \texttt{UNR} \rrbracket$? Our key idea is not to convert an expression directly to a corresponding logical term, but form equality with $\llbracket . \rrbracket_f$ recursively (defined in Figure 9). The subscript $f$ in $\llbracket e \rrbracket_f$ denotes the expression $e$. Moreover, we perform neither $\eta$-expansion (which does not preserve semantics in the presence of non-total terms) nor $\lambda$-lifting, and yet we allow arbitrary forms of recursive functions. We have such flexibility because we convert $\lambda$-abstraction and partial application directly with the help of `apply`. (Note that our logicization $\llbracket . \rrbracket_f$ can also produce higher-order logic formula for interactive proving by replacing $(\texttt{apply}(f, x))$ with $(f(x))$ and not converting the types.) No logicization work in the literature (including [6, 9, 28, 34]) deal with non-total terms. The work [6] uses approaches in [9, 28] 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:

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

Letting $x_0$ 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 [39].

**Theorem 5** (Logicization for axioms)**.** *Given closed definition $f = e^\tau$, the logical fomula $\exists f : \tau, \llbracket e \rrbracket_f$ is valid.*

$$
\begin{array}{rcl}
\oplus \in [+, -, *, /] & & \odot \in [>, <, =] \\
\llbracket . \rrbracket_f & : & \textbf{Expression} \to \textbf{Formula} \\
\llbracket \texttt{let (rec)}\ f = e \rrbracket_f & = & \llbracket e \rrbracket_f \quad \text{top-level defn} \\
\llbracket \texttt{BAD}^l \rrbracket_f & = & \begin{cases} \mathit{true} & \text{for axioms} \\ \mathit{false} & \text{for goals} \end{cases} \\
\llbracket \texttt{UNR}^l \rrbracket_f & = & \mathit{false} \\
\llbracket x \rrbracket_f & = & f = x \\
\llbracket n \rrbracket_f & = & f = n \\
\llbracket e_1^{\tau_1} \oplus e_2^{\tau_2} \rrbracket_f & = & \exists x_1 : \llbracket \tau_1 \rrbracket,\ \exists x_2 : \llbracket \tau_2 \rrbracket, \\
 & & \llbracket e_1 \rrbracket_{x_1} \wedge \llbracket e_2 \rrbracket_{x_2} \wedge f = x_1 \oplus x_2 \\
\llbracket e_1^{\tau_1} \odot e_2^{\tau_2} \rrbracket_f & = & \exists x_1 : \llbracket \tau_1 \rrbracket,\ \llbracket e_1 \rrbracket_{x_1} \wedge \\
 & & \exists x_2 : \llbracket \tau_2 \rrbracket,\ \llbracket e_2 \rrbracket_{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})) \\
\llbracket \lambda x^\tau . e \rrbracket_f & = & \forall x : \llbracket \tau \rrbracket,\ \llbracket e \rrbracket_{(\texttt{apply}(f,x))} \\
\llbracket \texttt{let}\ x^\tau = e_1\ \texttt{in}\ e_2 \rrbracket_f & = & \exists x : \llbracket \tau \rrbracket,\ \llbracket e_1 \rrbracket_x \ \wedge \llbracket e_2 \rrbracket_f \\
\llbracket e_1^{\tau_1}\ e_2^{\tau_2} \rrbracket_f & = & \exists x_1 : \llbracket \tau_1 \rrbracket,\ \llbracket e_1 \rrbracket_{x_1} \wedge \\
 & & \exists x_2 : \llbracket \tau_2 \rrbracket,\ \llbracket e_2 \rrbracket_{x_2} \wedge \\
 & & f = \texttt{apply}(x_1, x_2) \\
\llbracket K\ e_1^{\tau_1} \dots e_n^{\tau_n} \rrbracket_f & = & \exists x_1 : \llbracket \tau_1 \rrbracket,\ \llbracket e_1 \rrbracket_{x_1} \wedge \dots \wedge \\
 & & \exists x_n : \llbracket \tau_n \rrbracket,\ \llbracket e_n \rrbracket_{x_n} \wedge \\
 & & f = K\ (y_1, \dots, y_n) \\
\left\llbracket \begin{array}{c} \texttt{match}\ e_0^{\tau_0}\ \texttt{with} \\ \overrightarrow{K\ \overrightarrow{x^\tau} \to e} \end{array} \right\rrbracket_f & = & \begin{array}{l} \exists x_0 : \llbracket \tau_0 \rrbracket,\ \llbracket e_0 \rrbracket_{x_0} \wedge \\ \overrightarrow{(\bigwedge \forall \overrightarrow{x : \llbracket \tau \rrbracket}, (x_0 = K\ \overrightarrow{x}) \Rightarrow \llbracket e \rrbracket_f)} \end{array}
\end{array}
$$

**Figure 9.** Convert expression to logical formula

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, the goal `app_1` states that the ctx-info `v1=nil`, which is from the pattern matching `match` $v_1$ `with` $\{[] \to \dots\}$, implies the scrutinee. By [S-lam] and [R-s-save], we have $\mathcal{L} = \forall v_1 : \texttt{'a list}, \forall v_2 : \texttt{'a list}, v_1 = \texttt{nil}$. The scrutinee is $\llbracket \texttt{len}\ v_2 = \texttt{len}\ v_1 + \texttt{len}\ v_2 \rrbracket_{\texttt{true}}$. 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.

**Theorem 6** (Logicization for goals: validity preservation)**.** *For all (possibly open) expression $e^\tau$, for all $fv(e)$, $\exists f : \tau$, if $\llbracket e \rrbracket_f$ is valid and $e \to e'$ for some $e'$, then $\llbracket e' \rrbracket_f$ is valid.*

More details on design choices are in [39]. Here, we highlight a few. (1) Only functions called in contracts are converted to Alt-ergo axioms. (2) In Figure 9, there are two conversions for `BAD`, *true* for axioms. This is for generating a harmless axiom *true* for the crashing branch of a partial function called in contracts. (3) For goals, the $\llbracket e \rrbracket_f$ collects ctx-info *before* a scrutinee of a `match` expression, thus, $\llbracket \texttt{BAD} \rrbracket_f = \llbracket \texttt{UNR} \rrbracket_f = \mathit{false}$, which implies everything, which is what we want.

### 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), the SL machine does not violate call-by-value execution.

One might worry that the rule [S-m-match] causes exponential code explosion for static analysis (although no run-time overhead). From our current observation, quite often the scrutinee is `if` $b$ `then` $d$ `else` $e$ where $e$ is `BAD` or `UNR`. As blames trigger the SL machine to immediately rebuild the blame with the rest of the stack, applying the rule [S-m-match], we do not have duplication but have a desired smaller formula for the SMT solver. We have implemented a prototype 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 a quad-core 2.93GHz CPU and 3.2GB memory. We take some examples from [26] and OCaml stdlib and time the static checking. The column Ann gives the LOC count for contract annotations. One advantage of the SL

**Table 1.** Results of preliminary experiments

| program | total LOC | Ann LOC | Time (sec) |
|---|---|---|---|
| intro123, neg, mc91 | 28 | 5 | 0.10 |
| ack, fhnhn, zipunzip | 25 | 4 | 0.16 |
| arith, sum, max | 26 | 4 | 0.20 |
| OCaml stdlib/list.ml | 81 | 16 | 0.72 |

machine is that it allows rules to be easily added or removed. This paper focuses on the theory of hybrid contract checking. We leave optimization and rigorous experimentation on tuning the strength of symbolic simplification and the frequency of calling an SMT solver as future work.

## 6. Hybrid contract checking

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

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

We have:

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

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

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

In [HCC], the residual code $f_i\sharp$'s parameter “?” waits for a caller's name. For example, *if* an SMT solver cannot prove the goal `app_2` in §5.2 (although it can), recalling R3 in §5.2, the residual code `append`$\sharp$ is:

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

which says that we only have to check the 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, 40])**.** *For all expression $e$, total contract $t$, blames $r_1, r_2, r_3, r_4$,* $\quad (e \overset{r_1}{\underset{r_2}{\bowtie}} t) \overset{r_3}{\underset{r_4}{\bowtie}} t = e \overset{r_1}{\underset{r_4}{\bowtie}} t.$

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

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

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. See [39] for more details.

## 7. Related work

Contract semantics were first formalized for a strict language [7, 11] and later for a lazy language [40]. This paper adapts and re-formalizes some of their ideas on contract satisfaction and contract checking. Detailed design difference is explained in §4.

Pre/post-condition specification using logical formulae [2, 15, 17, 34] allows programmers to existentially quantify over infinite domains or express metaproperties that are not expressible in contracts. We like the idea of ghost refinement [36], which separates properties that can be converted to program code from the metaproperties expressed only as logical formulae. As there are always limits to automatic static checking, it is practical to convert some difficult checks to dynamic checks. Unlike pre/post-condition specification, refinement types and contracts allow us to study sub-contract relations [11, 41], recursive contracts [7], and polymorphic contracts [3]. Contracts also enjoy interesting mathematical properties [7, 11, 39, 40].

One might recall hybrid refinement type checking (HTC) [13, 24]. In theory [16], (picky, 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 [36] invents a Kind checker to report ill-formed refinement types. As discussed in §4.2, we check $e \rhd t$ for crash-freeness 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 a similar purpose as the typing environment in HTC. But symbolic simplification gives more flexibility in such ways as teasing out the path sensitivity analysis with the rule [S-m-match], etc. We hope this work opens a venue to compare HCC and HTC in practice. Notably, VeriFast [20] (for verifying C and Java code) suggests that symbolic execution is faster than the verification condition generation method [2, 15].

Khoo et al. [23] mix type checking and symbolic execution. Besides they do not generate residual code, they require 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. Moreover, their symbolic expression is given in linear arithmetic, which is more restrictive than ours.

Our approach is different from [36], 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. Theirs has a security motivation.

Some work [25, 26, 32, 33] suggests converting programs to a higher-order recursive scheme (HORS), which generates (possibly infinite) trees, and specify properties in the form of a trivial automaton and do model checking to see whether HORS satisfies its desired property. Our approach is completely different although we both do reachability checking. They work on automata while we work on programs directly. Our approach is *modular* while theirs is not. They deal with local `let rec` while we do not, but we could infer local contract with method in [21] or inline the local let rec function for a fixed number of times. They deal with protocol checking while we do not, except where a protocol checking problem can be converted to checking the reachability of `BAD`.

The contextual information synthesis and conversion of expression to logical formula is inspired by the use of the application $\bullet$ in [18, 19], which makes conversion of higher-order functions easier. But we use the technique in different contexts.

Many papers on program verification [2, 10, 15, 30, 31, 37] focus on memory leaks, array bound checks, etc. and a few handle higher-order functions and recursive predicates. Our work focuses on more advanced properties and precise blaming of functions at fault. Contract checking in the imperative world is lead by [10], which statically checks contract satisfaction at the bytecode CIL level and runs dynamic checking separately. Residualization has not been done in [10]. We may adapt some ideas in [20] 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 the SL machine, our approach gives precise blame at both compile-time and run-time in the presence of higher-order functions. In the 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.

## Acknowledgments

I would like to thank Xavier Leroy, Francois Pottier, Nicolas Pouillard, Martin Berger, Simon Peyton Jones, Michael Greenberg, and the anonymous reviewers for their comments.

## 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. Unpublished, 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] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS: the Intl. Conf. on Formal Verf. of OO Software*, pages 10–30, 2010.

[11] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Functional and Logic Prog.*, pages 226–241, 2006.

[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP: the ACM SIGPLAN Intl. Conf. on Fnl. Prog.*, pages 48–59, 2002.

[13] C. Flanagan. Hybrid type checking. In *POPL: the ACM SIGPLAN-SIGACT symp. on Prin. of Prog. Lang.*, pages 245–256, 2006.

[14] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI: the ACM SIGPLAN conf. on Prog. Lang. Design and Impl.*, pages 237–247. ACM, 1993.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[16] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL: the ACM SIGPLAN-SIGACT symp. on Prin. of Prog. Lang.*, pages 353–364, 2010.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[18] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *PPDP: the ACM SIGPLAN intl. conf. on Prin. and practice of Decl. Prog.*, pages 191–202, 2004.

[19] K. Honda, M. Berger, and N. Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *ICALP: the Intl. Colloq. on Autamata, Lang. and Prog.*, pages 360–371, 2006.

[20] 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.

[21] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV: the intl. conf. on Comp. Aided Verf.*, pages 262–274, 2011.

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

[23] Y. P. Khoo, B.-Y. E. Chang, and J. S. Foster. Mixing type checking and symbolic execution. In *PLDI: the ACM SIGPLAN conf. on Prog. Lang. Design and Impl.*, pages 436–447, 2010.

[24] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32:6:1–6:34, February 2010.

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

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

[27] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

[28] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS: the Intl. Conf. on Tools and Algo. for the Construction and Anls. of Syst.*, pages 312–327, 2010.

[29] B. Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[30] M. Might. Logic-flow analysis of higher-order programs. In *POPL: the ACM SIGPLAN-SIGACT symp. on Prin. of Prog. Lang.*, pages 185–198, 2007.

[31] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP*, pages 62–73, 2006.

[32] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS: IEEE Symp. on Logic in Computer Science*, pages 81–90, 2006.

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

[34] Y. Régis-Gianas and F. Pottier. A hoare logic for call-by-value functional programs. In *MPC*, pages 305–335, 2008.

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

[36] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 15–27, 2011.

[37] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL: the ACM SIGPLAN-SIGACT symp. on Prin. of Prog. Lang.*, pages 214–227, 1999.

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

[39] D. N. Xu. Hybrid contract checking via symbolic simplification. INRIA technical report RR-7794, 2011.

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

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