Hybrid Contract Checking via Symbolic Simplification

Na Xu

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Hybrid Contract Checking via Symbolic Simplification

Dana N. Xu

N° 7794

Novembre 2011

Programs, Verification and Proofs

ISSN 0249-6399 ISRN INRIA/RR--7794--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Hybrid Contract Checking via Symbolic Simplification

Dana N. Xu

Theme : Programs, Verification and Proofs
Algorithmics, Programming, Software and Architecture
Équipes-Projets Gallium

Rapport de recherche n° 7794 — Novembre 2011 — 69 pages

Abstract: Program errors are hard to detect and to prove absent. Contract checking allows us to (a) statically verify that a function satisfies its contract; (b) precisely blame functions at fault both statically and dynamically when there is a contract violation. Static contract checking catches all bugs but can only check restricted properties while dynamic checking can check more expressive properties, but is not complete. In this paper, we integrate static and dynamic contract checking for a subset of OCaml. We exploit a static checker as much as possible and leave the residual contract satisfaction checks to run-time. Thus, no (potential) bugs can escape and yet expressive properties can be expressed.

Key-words: contract semantics, static, dynamic, hybrid, contract checking, functional language, verification, debugging

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Vérification de contrats hybride par simplification symbolique

Résumé : Il est difficile de détecter des erreurs dans des programmes, ou de démontrer leur absence. Permettre aux programmeurs d'écrire des spécifications formelles et précises, en particulier sous la forme de contrats, est une approche commune pour vérifier des programmes et trouver des erreurs. Nous formalisons et proposons une implémentation d'un vérificateur hybride de contrats pour un sous-ensemble d'OCaml. La technique principale que nous mettons en œuvre est la simplification symbolique, qui permet de combiner facilement les vérifications statiques et dynamiques de contrats. La technique que nous proposons consiste à vérifier qu'une fonction satisfait son contrat ou indique quelle est la fonction à l'origine de sa violation. Quand la satisfaction d'un contrat n'est pas décidable statiquement, du code de test est ajouté au programme afin d'effectuer les vérifications à l'exécution.

Mots-clés : la sémantique du contrat, statique, dynamique, hybride, langage fonctionnel, vérification, débogage

Contents

1 Introduction

Constructing reliable software is difficult even with functional languages. Formulating and checking (statically or dynamically) logical assertions [18, 15, 2, 5, 35], especially in the form of contracts [28, 13, 7, 14, 39], 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 run-time 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 $\forall g : \texttt{int} \to \texttt{int},\ (g\ 1) \geq 0 \Rightarrow (g\ 1) - 1 \geq 0$ 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 [39], 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, 39].

- We define $e \in t$ and $e \triangleright t$ and prove a theorem "$e \triangleright t$ is crash-free $\iff$ $e \in t$" (§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 [40] 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 [20, 19] 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 [28, 13]. 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 $t_{\texttt{inc}}$:

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

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:

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

In the upper box, the argument of `inc` is guarded by the check $x > 0$; in the lower box, the result of `inc` is guarded by the check $y > 0$. If a check succeeds, the

original term is returned; otherwise, the special constructor `BAD` is reached and a blame is raised. In this case, `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, 39] 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:

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

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

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

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

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

Unlike [37] 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:

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

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:

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

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 $\texttt{f1} \triangleright t_{\texttt{f1}}$ is $(\lambda g.(g\ 1) - 1) \triangleright (\{x \mid x \geq 0\} \to \{y \mid y \geq 0\}) \to \{z \mid z \geq 0\}$, which expands to:

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

After applying some conventional simplification rules, we have:

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

We see that the inner $\texttt{BAD}^{(4,5,\text{“}f_1\text{”})}$ has been simplified away, because $x = x_2 = 1$ and $(\texttt{if } 1 \geq 0 \texttt{ then } 1 \texttt{ else BAD}^{(4,5,\text{“}f_1\text{”})})$ 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, $\texttt{f1} \triangleright t_{\texttt{f1}}$ checks whether `f1` satisfies its postcondition by assuming its precondition holds. At each call site of `f1`, we wrap the function with $\triangleleft$. 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 $\texttt{zut} \triangleleft \{x \mid \texttt{true}\} = \texttt{zut}$, we then have the term $\texttt{f3} \triangleright t_{\texttt{f3}}$ to be:

$$((\texttt{f1} \triangleleft t_{\texttt{f1}})\ \texttt{zut}) \triangleright \{v \mid v > 0\}$$

which *requires* `f3` to satisfy `f1`'s precondition and assumes `f1` satisfies its postcondition because $\texttt{f1} \triangleright t_{\texttt{f1}}$ 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:

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

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):

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

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 `UNR`s 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:

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

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

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

pgm	::=	$def_1, \ldots, def_n$	**Program**
τ	::=	$\overrightarrow{\tau}\ T \mid \tau_1 \to \tau_2$	**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	$\in$	**Exp**	**Expressions**
a, e, p	::=	n	integers
	$\mid$	r	blame
	$\mid$	$x \mid \lambda(x^\tau).e \mid e_1\ e_2$	
	$\mid$	$\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{alt}$	pattern-matching
	$\mid$	$K\ \overrightarrow{e}$	constructor

r	::=	$\texttt{BAD}^l \mid \texttt{UNR}^l$	**Blames**
l	::=	$(n_1, n_2, \texttt{String})$	**Label**
alt	::=	$K\ (x_1^{\tau_1}, \ldots, x_n^{\tau_n})\ \to\ e$	**Alternatives**
val	::=	$n \mid x \mid r \mid K\ \overrightarrow{v} \mid \lambda(x^\tau).e$	**Values**

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. 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`. 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 good enough to infer a contract for a local function.) 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` {`true` $\rightarrow e_1$; `false` $\rightarrow e_2$}.

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 superscript, e.g. e^τ or t^τ) whenever necessary.

The two contract exceptions (also called blames) $\texttt{BAD}^l$ and $\texttt{UNR}^l$ are adapted from [39]. They are for internal usage, not visible to programmers. The label l contains information such as function name and source code 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 $\texttt{BAD}^{\texttt{head}}$.

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, $x\!:\!\{y \mid y > 0\} \rightarrow \{z \mid z > x\}$ 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, $(x\!:\!\{y \mid y > 0\}, \{z \mid z > x\})$ whose short hand is $(\{x \mid x > 0\}, \{z \mid z > x\})$. Contract `Any` is a universal contract that any expression satisfies. We support higher order contracts, e.g. $k\!:\!(\{x \mid x > 0\} \rightarrow \{y \mid y > x\}) \rightarrow \{z \mid k\ 5 > -1\}$ for a function `let f g = g 2`.

3.2 Type checking rules for expression

The language M is statically typed in the conventional way. Figure 2 gives type checking rules. A type judgement has the form

$$\Gamma \vdash e^\tau$$

which states that given Γ (which is a mapping from variable to its type), e has type τ assuming that any free variable in it has type given by Γ. If $\Gamma = \emptyset$, we omit the Γ, and write $\vdash e^{\tau}$.

$$\Gamma \vdash \texttt{BAD} :: \tau \quad \text{[T-BAD]} \qquad \Gamma \vdash \texttt{UNR} :: \tau \quad \text{[T-UNR]}$$

$$\frac{v :: \tau \in \Gamma}{\Gamma \vdash v :: \tau} \quad \text{[T-Var]} \qquad \frac{\Gamma, x :: \tau_1 \vdash e :: \tau_2}{\Gamma \vdash (\lambda(x^{\tau_1}).e) :: \tau_1 \to \tau_2} \quad \text{[T-Lam]}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2} \quad \text{[T-App]}$$

$$\frac{K :: \overrightarrow{\tau} \to T \in \Gamma \qquad \Gamma \vdash \overrightarrow{e} :: \overrightarrow{\tau}}{\Gamma \vdash K\ \overrightarrow{e} :: T\ \overrightarrow{\alpha}} \quad \text{[T-Con]}$$

$$\frac{\Gamma \vdash e_0 :: T\ \overrightarrow{\tau} \qquad \Gamma, \{v :: T\ \overrightarrow{\tau}\}, \{\overrightarrow{K_i\ \overrightarrow{x_i} :: T\ \overrightarrow{\tau}}\} \vdash e_i :: \tau}{\Gamma \vdash (\texttt{case}\ e_0\ \texttt{of}\ (v^{T\ \overrightarrow{\tau}})\ \{K_i\ \overrightarrow{x_i} \to e_i\}) :: \tau} \quad \text{[T-Match]}$$

Figure 2: Type Checking Rules

As we do type checking before contract checking, we assume all expressions are well-typed (i.e. no type error) in the rest of this paper. Note that nothing substantial in the paper depends delicately on the type system. The reason we ask that programs are well-typed is to avoid the technical inconvenience in designing the semantics of contracts if, say, evaluation finds an ill-typed expression `(3 True)`.

3.3 Operational semantics

The semantics of our language is given by reduction rules in Figure 3. For a top-level function, we fetch its definition from the evaluation envrionment Δ. We adapt some basic definitions from [39]. Definition 1 defines the usual contextual equivalence. Two expressions are said to be semantically equivalent, 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, $\mathcal{C}[\![e_1]\!] \to^* r \iff \mathcal{C}[\![e_2]\!] \to^* r$*

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

Definition 2 (Diverges)**.** *A closed expression e* diverges*, written $e\uparrow$, iff either $e \to^* \texttt{UNR}$, or there is no value val such that $e \to^* val$.*

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

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

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

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

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

Figure 3: Semantics of the language M

3.4 Crashing

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

Definition 3 (Crash)**.** *A closed term* e crashes *iff* $e \to^* \texttt{BAD}$.

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* $\texttt{BAD} \not\in_s e$. *Similarly, a context* $\mathcal{C}$ *is syntactically safe iff* $\texttt{BAD} \not\in_s \mathcal{C}$.

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

Definition 5 (Crash-free expression)**.** *A (possibly-open) expression* e *is crash-free iff : for all* $\mathcal{C}$ *such that* $\texttt{BAD} \not\in_s \mathcal{C}$ *and* $\vdash \mathcal{C}[\![e]\!] :: \texttt{bool}$, $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$.

The notation $\vdash \mathcal{C}[\![e]\!] :: \texttt{bool}$ 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. $\lambda x.\texttt{if}\ x * x \geq 0\ \texttt{then}\ x + 1\ \texttt{else BAD}$.

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

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

Proof. Since there is no BAD syntactically in e, for all context $\mathcal{C}$, such that there is no BAD syntatically in $\mathcal{C}$, then $\mathcal{C}[\![e]\!] \not\to^* \texttt{BAD}$. By definition 5 (Crash-free expression), e 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.

Lemma 2 (Neutering)**.** *If* e *is crash-free, then* $\lfloor e \rfloor \equiv_s e$.

Proof. Since e is crash-free, all BADs in e are not reachable so by converting all BADs in e to UNR by $\lfloor . \rfloor$ does not change the semantics of e. Formally, we prove this by induction on reduction rules. □

Lemma 3 (Crash-free Preservation)**.** *Given* $e_1 \to e_2$*,*

$$e_1 \textit{ is crash-free} \iff e_2 \textit{ is crash-free}$$

Proof. We prove two directions by contradiction.
($\Rightarrow$)
Suppose e_2 is not crash-free. By Definition 5^{p12} (Crash-free Expression), there exists a $\mathcal{C}$ such that $\texttt{BAD} \notin_s \mathcal{C}$ and $\mathcal{C}[\![e_2]\!] \to^* \texttt{BAD}$. By [E-ctx] and $e_1 \to e_2$ and $\mathcal{C}[\![e_2]\!] \to^* \texttt{BAD}$, we have: $\mathcal{C}[\![e_1]\!] \to^* \mathcal{C}[\![e_2]\!] \to^* \texttt{BAD}$. As we know e_1 is crash-free, we reach contradiction. Thus, we are done.
($\Leftarrow$)
Suppose e_1 is not crash-free. By Definition 5^{p12} (Crash-free Expression), there exists a $\mathcal{C}$ such that $\texttt{BAD} \notin_s \mathcal{C}$ and $\mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$. By [E-ctx] and $e_1 \to e_2$ and confluence of the language, we have $\mathcal{C}[\![e_2]\!] \to^* \texttt{BAD}$. With the assumption that e_2 is crash-free, we reach contradiction. Thus, we are done. □

Lemma 4 (Crash-free function)**.** *For all (possibly-open) terms* $\lambda x.e$*,*

$$\lambda x.e \textit{ is crash-free}$$
$$\iff$$
$$\textit{for all (possibly-open) crash-free } e',\ e[e'/x] \textit{ is crash-free.}$$

Proof. We prove two directions separately.
($\Rightarrow$)

$$\begin{array}{ll} & \lambda x.e \text{ is crash-free} \\ \Rightarrow & \text{(By Lemma } 2^{p12}\text{, } e' \text{ is crash-free } \Rightarrow \lfloor e' \rfloor \equiv_s e' \\ & \text{and by the definition of crash-free expression)} \\ & \text{for all crash-free } e',\ e[e'/x] \text{ is crash-free} \end{array}$$

($\Leftarrow$) We have the following proof.

$$\begin{array}{ll} & \text{for all } \mathbf{cf}\ e', e[e'/x] \text{ is crash-free} \\ \iff & \text{(By Lemma } 3^{p13}\text{)} \\ & \text{for all } \mathbf{cf}\ e', (\lambda x.e)\ e' \text{ is crash-free} \\ \iff & \text{(By Definition } 5^{p12} \text{ (Crash-free Expression))} \\ & \text{for all } \mathbf{cf}\ e', \forall \mathcal{C},\ \texttt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![(\lambda x.e)\ e']\!] \not\to^* \texttt{BAD} \\ \Rightarrow & \text{(By Lemma } 2^{p12}\text{, } e' \text{ is crash-free } \Rightarrow \lfloor e' \rfloor \equiv_s e') \\ & \forall \mathcal{C},\ \texttt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![(\lambda x.e)\ \lfloor e' \rfloor]\!] \not\to^* \texttt{BAD} \\ \Rightarrow & \text{(By } \texttt{BAD} \notin_s \lfloor e' \rfloor\text{)} \\ & \forall \mathcal{C},\ \texttt{BAD} \notin_s \mathcal{C},\ \mathcal{C}[\![(\lambda x.e)]\!] \not\to^* \texttt{BAD} \\ \iff & \text{(By Definition } 5^{p12} \text{ (Crash-free Expression))} \\ & \lambda x.e \text{ is crash-free} \end{array}$$

□

3.5 Behaves-the-same

We define an ordering, named *Behaves-the-same*, which is useful in later sections.

Definition 6 (Behaves the same)**.** *Expression e_1 behaves the same as e_2 w.r.t. a set of exceptions R, written $e_1 \ll_R e_2$, iff for all contexts $\mathcal{C}$, such that $\forall i \in \{1,2\}$. $\vdash \mathcal{C}[\![e_i]\!] :: \texttt{bool}$*

$$\mathcal{C}[\![e_2]\!] \rightarrow^* r \in R \quad \Rightarrow \quad \mathcal{C}[\![e_1]\!] \rightarrow^* r$$

Definition 6^{p14} says that e_1 either behaves the same as e_2 or throws an exception from R. (The definition does not look as strong as that, but as every theorist knows, it is. For example, could e_1 produce `true` while e_2 produces `false`? No, because we could find a context $\mathcal{C}$ that would make $\mathcal{C}[\![e_2]\!]$ throw an exception while $\mathcal{C}[\![e_1]\!]$ does not.) In our framework, there are only two exceptional values in R: `BAD` and `UNR`. Certainly, if e_2 itself throws an exception, then e_1 must throw the same exception.

As we only have two exceptional values `BAD`, `UNR` (which are dual to each other) in R, this yields Lemma 5^{p14}. We omit $\{\}$ if there is only one element in R.

Lemma 5 (Properties of Behaves-the-same)**.** *For all closed e_1 and e_2,*

$$e_1 \ll_{\texttt{UNR}} e_2 \iff e_2 \ll_{\texttt{BAD}} e_1$$

Proof. We prove two directions separately.

($\Rightarrow$) We have the following proof:

$$\begin{array}{ll} & e_1 \ll_{\texttt{UNR}} e_2 \\ \iff & \text{(By defn of } \ll_{\texttt{UNR}}) \\ & \forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{UNR} \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{UNR} \\ \iff & \text{(By logic)} \\ & \forall \mathcal{C}.\ \mathcal{C}[\![e_1]\!] \not\rightarrow^* \texttt{UNR} \Rightarrow \mathcal{C}[\![e_2]\!] \not\rightarrow^* \texttt{UNR} \end{array}$$

We want to show that $\forall \mathcal{D}.\ \mathcal{D}[\![e_1]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{D}[\![e_2]\!] \rightarrow^* \texttt{BAD}$.

Assume $\mathcal{D}[\![e_1]\!] \rightarrow^* \texttt{BAD}$.

Let $\mathcal{C} = \texttt{match } (\mathcal{D}[\![\bullet]\!]) \texttt{ with } \{\texttt{DEFAULT} \rightarrow \texttt{UNR}\}$

Now we have $\mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_2]\!] \not\rightarrow^* \texttt{UNR}$.

Since $\mathcal{C}[\![e_2]\!] = \texttt{case } \mathcal{D}[\![e_2]\!] \texttt{ with } \{\texttt{DEFAULT} \rightarrow \texttt{UNR}\}$, we have $\mathcal{D}[\![e_2]\!] \rightarrow^* \texttt{BAD}$.

So we have

$$\forall \mathcal{D}.\ \mathcal{D}[\![e_1]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{D}[\![e_2]\!] \rightarrow^* \texttt{BAD}$$

($\Leftarrow$) By replacing `BAD` by `UNR` and `UNR` by `BAD` in the above proof for the direction ($\Rightarrow$), we get the proof for the direction ($\Leftarrow$). □

3.6 Crashes-more-often

We study a specialized ordering *crashes-more-often*, which plays a crucial role in proving our main theorems.

Definition 7 (Crashes-more-often)**.** *An expression e_1 crashes more often than e_2, written $e_1 \preceq e_2$, iff $e_1 \ll_{\texttt{BAD}} e_2$.*

Informally, e_1 crashes more often than e_2 if they behave in exactly the same way except that e_1 may crash when e_2 does not. By Definition 7^{p14}, Lemma 5^{p14} also says that:

$$e_1 \ll_{\text{UNR}} e_2 \iff e_2 \preceq e_1$$

Theorem 1 (Crashes-more-often is AntiSymmetric)**.** *For all expressions e_1 and e_2, $e_1 \preceq e_2$ and $e_2 \preceq e_1$ iff $e_1 \equiv_s e_2$.*

Proof. It follows immediately from the definition of $\equiv_s$ (Definition 1^{p11}) and the definition of $\preceq$. □

The crashes-more-often operator has many properties. Lemma 6^{p15} says that BAD crashes-more-often then all expressions; all expressions crash more often then a diverging expression. Lemma 7^{p15} gives more intuitive properties.

Lemma 6 (Properties of Crashes-more-often - I)**.**

$$\begin{array}{ll} (a) & \texttt{BAD} \preceq e_2 \\ (b) & e_1 \preceq e_2 \quad \textit{if } e_2\uparrow \end{array}$$

Proof. We prove each property separately (all by contradiction) and we assume type soundness.

(a) Assume there exists a context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$ and $\mathcal{C}[\![\texttt{BAD}]\!] \not\rightarrow^* \texttt{BAD}$. There are two possibilities for $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$: (1) the BAD is from the context $\mathcal{C}$; (2) the BAD is from the hole e_2. For case (1), we must have $\mathcal{C}[\![\texttt{BAD}]\!] \rightarrow^* \texttt{BAD}$ since we use the same context $\mathcal{C}$. For case (2), if the hole is evaluated, we reach BAD immediately. So we reach a contradiction and we are done.

(b) Given $e_2\uparrow$, assume there exists a context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$ and $\mathcal{C}[\![e_1]\!] \not\rightarrow^* \texttt{BAD}$. Since $e_2\uparrow$ and $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$, we know the BAD is from the context $\mathcal{C}$. So no matter what e_1 is, we have $\mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}$. Thus, we again reach a contradiction and we are done.

□

Lemma 7 (Properties of Crashes-more-often - II)**.** *If $e_1 \preceq e_2$*

$$\begin{array}{lrcl} (a) & e_1 \rightarrow^* K\ \overline{f_1} & \Rightarrow & e_2 \rightarrow^* K\ \overline{f_2} \textit{ or } e_2\uparrow \\ (b) & e_1\uparrow & \Rightarrow & e_2\uparrow \\ (c) & e_1 \textit{ is crash-free} & \Rightarrow & e_2 \textit{ is crash-free} \\ (d) & e_1 \rightarrow^* \lambda x.e_1' & \Rightarrow & e_2 \rightarrow^* \lambda x.e_2' \textit{ or } e_2\uparrow \end{array}$$

Proof. We prove each property separately (all by contradiction):

(a) Given $e_1 \rightarrow^* K\ \overline{f_1}$, assume neither $e_2 \rightarrow^* K\ \overline{f_2}$ nor $e_2\uparrow$. Then we must have $e_2 \rightarrow^* \texttt{BAD}$. By the definition of $\preceq$ and the fact that $e_1 \preceq e_2$, if $e_2 \rightarrow^* \texttt{BAD}$, then $e_1 \rightarrow^* \texttt{BAD}$. Since $e_1 \rightarrow^* K\ \overline{f_1}$, we reach a contradiction and we are done.

(b) Given $e_1\uparrow$, assume $e_2 \not\downarrow$. Then $e_2 \rightarrow^* \mathit{val}$ and there exists a syntactically safe context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. But $\mathcal{C}[\![e_1]\!]$ always diverges as e_1 diverges if $\texttt{BAD} \notin_s \mathcal{C}$. By the fact that $e_1 \preceq e_2$ and by the definition of $\preceq$, we reach a contradiction and we are done.

(c) Given e_1 is crash-free, assume e_2 is not crash-free. By Definition 5^{p12} (Crash-free Expression), there exists a syntactically safe context $\mathcal{C}$ such that $\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD}$. By the fact that $e_1 \preceq e_2$ and by the definition of $\preceq$, we have $\mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}$. This contradicts with another assumption that e_1 is crash-free. Since we reach a contradiction, we are done.

(d) The proof is similar to that in (a).

□

4 Contracts

Findler and Felleisen (FF) first introduced an algorithm for dynamic higher order contract checking [13]. Blume and McAllester [7] then define a semantics for contract satisfaction and show its sound-and-completeness with respect to the FF-algorithm. As the algorithm and the contract semantics are defined by two groups of people, there are some mismatches addressed in [12]. Later, [39] defines both a contract semantics and a (static) checking algorithm for a lazy language. In this paper, we follow the style in [39], design contract satisfaction and checking algorithm for a strict language. As diverging contracts make dynamic contract checking unsound (explained in Section 4.5) and we do hybrid checking, we focus on total contracts.

Definition 8 (Total contract)**.** *A contract t is total iff*

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

Our definition of total contract is different from that in [7], but close to the crash-free contract in [39] with an additional condition that $\lambda x.p$ is a terminating function. For example, contract $\{x \mid x \neq \texttt{[]}\} \rightarrow \{y \mid \texttt{head}\ x > y\}$ is total in our framework because `head` x does not crash for all x satisfying $\{x \mid x \neq \texttt{[]}\}$. Such a contract is not total in [7] because a crashing function `head` is called in a predicate contract.

4.1 Type checking for contracts

A contract type judgement has the form

$$\Gamma \vdash_c t \in \tau$$

which states that given Γ (a mapping from program variable to its type, and from type variable α to its kind k), e has type τ assuming that any free variable in it has type given by Γ. Contract type checking rules are shown in Figure 4.

$$\frac{\Gamma, \alpha :: k \vdash_c t :: \tau}{\Gamma \vdash_c (\forall \alpha :: k.\ t) :: \tau} \quad \text{[C-FORALL]}$$

$$\frac{}{\Gamma \vdash_c \texttt{Any} :: \tau} \quad \text{[C-ANY]} \qquad \frac{\Gamma, x :: \tau \vdash_c e :: \texttt{Bool}}{\Gamma \vdash_c \{x \mid e\} :: \tau} \quad \text{[C-ONE]}$$

$$\frac{\Gamma \vdash_c t_1 :: \tau_1 \qquad \Gamma, x :: \tau_1 \vdash_c t_2 :: \tau_2}{\Gamma \vdash_c x : t_1 \to t_2 :: \tau_1 \to \tau_2} \quad \text{[C-FUN]}$$

$$\frac{\Gamma \vdash_c t_1 :: \tau_i \qquad \Gamma, x :: \tau_1 \vdash_c t_2 :: \tau_2}{\Gamma \vdash_c (x : t_1, t_2) :: (\tau_1, \tau_2)} \quad \text{[C-TUPLE]}$$

Figure 4: Type Checking Rules for Contract

For a well-typed expression e, define $e \in t$ thus:

$$e \in \{x \mid p\} \iff e\uparrow \text{ or } (e \text{ is crash-free and } p[e/x] \to^* \texttt{true}) \quad \text{[A1]}$$

$$e \in x\colon t_1 \to t_2 \iff e\uparrow \text{ or } (e \to^* \lambda x.e_2 \text{ and } \forall val_1 \in t_1.\ (e\ val_1) \in t_2[val_1/x]) \quad \text{[A2]}$$

$$e \in (x\colon t_1, t_2) \iff e\uparrow \text{ or } (e \to^* (val_1, val_2) \text{ and } val_1 \in t_1 \text{ and } val_2[val_1/x] \in t_2[val_1/x]) \quad \text{[A3]}$$

$$e \in \texttt{Any} \iff \texttt{true} \quad \text{[A4]}$$

Figure 5: Contract Satisfaction

4.2 A semantics for contract satisfaction

We give the semantics of contracts by defining "e satisfies t" (written $e \in t$) in Figure 5 inspired by [7, 39]. 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 [39] is that, we do not allow $p[e/x]$ in [A1] to diverge while [39] allows because they only do static checking. We support dependent tuple contracts, that are not in [7, 39]. One difference from [7] is that, they say that a crashing expression does not satisifay 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. It is obvious that `Any` is useful in a lazy language [39] as we may want to ignore some subcomponents of a constructor. We explain why `Any` is also useful for a strict language in Section 4.6.

Definition 9 (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* $\forall e.\ e \in t_1 \Rightarrow e \in t_2$

4.3 The wrappers

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

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

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

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

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

Figure 6: Contract checking with the wrappers

As mentioned in Section 2, the essence of contract checking is the two wrappers $\triangleright$ and $\triangleleft$, which are dual to each other (defined in Figure 6). 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.

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]). The wrappers are defined such that Theorem 2 holds.

Theorem 2 (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 2 is in Appendix A, which is similar to that in [40]. In practice, we only need Thereom 3, i.e. one direction of Thereom 2.

Theorem 3 (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.

4.4 Open expressions and contracts

For open expressions, we use the same idea in [39]. Suppose the declared contracts for `f` and `g` are t_f, t_g respectively, and the definition of `g` is $\mathtt{g} = e_g$ where `f` is called in e_g. Then, instead of checking that $e_g \in t_g$, we check that

$$(\lambda f.\ e_g)\ \in\ t_f \to t_g$$

That means we simply lambda-abstract over any variables free in e_g. The same idea applies for the recursive functions. If the programmer specifies the contract t_f for a definition $\mathtt{f} = e$, then it suffices to check that

$$\lambda f.e\ \in\ t_f \to t_f$$

which is easier because $\lambda f.e$ does not call `f` recursively. There is nothing new here – it is just the standard technique of loop invariants in another guise – but it is packaged very conveniently.

In other words, imagine we have a contract judgement:

$$\Delta \vdash e \in t$$

which states that given Δ, which is a mapping from variable to its type, contract and definition.

Definition 10 (Contract judgement)**.** *We write $\Delta \vdash e \in t$ to mean that e has contract t assuming that any free variable in e has contract given by Δ and any free variable in t has definition given by Δ. Suppose $\Delta = \{f_1 \mapsto (\tau_1, t_1, e_1), \ldots, f_n \mapsto (\tau_n, t_n, e_n)\}$, we define:*

$$\Delta \vdash e \in t \iff \lambda f_1 \ldots . f_n.e \in t_1 \to \cdots \to t_n \to t$$

This means, in theory (i.e. in the formalization of the verification), we only need to deal with closed expressions; in practice (i.e. in the implementation), we may refer to the environment Δ when necessary. We can simply check crash-freeness of $e[(g \lhd t_g)/g] \rhd t_f[(g \lhd t_g)/g]$ where a call to g is replaced by $g \lhd t_g$. This idea holds for recursive calls of f in e as well, we check $e[f \lhd t_f/f] \rhd t_f$. (Note that f is not allowed to be used in t_f.)

4.5 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 []
```

`fb` $\triangleright t_{\texttt{fb}}$ is $\lambda x_1.((\lambda x.$`head [ ]`$)$ (`if loop` x_1 `then` x_1 `else BAD`$))$, which diverges whenever applied because of the `loop`. However, the function `fb` is not crash-free.

Consider the higher order function `f` in Section 4.3, one might wonder whether we have to check the argument of the higher order function `f` to be terminating because `k` is called in the contract. The answer is no. By inspecting [P1] and [P2], we can see that an argument is always evaluated earlier than the x in t_2. So we will not encounter the situation that a divergent contract hides a crash.

We only have to prove termination of functions used in contracts, not all the functions in a program. We can adapt ideas in [26, 34, 4] to build an efficient automatic termination checker.

4.6 Contract `Any`

There is a debate in [12] on whether it is useful to have a top ordering contract `Any`. We want `Any` because we want to give a function, that always fails, a contract to satisfy, so that we do not blame it at its definition site during SCC because $\forall e, e \triangleright$`Any` $=$ `UNR`, which is crash-free. Consider a popular OCaml library function:

```
contract failwith = {x | true} -> Any
let failwith str = raise (Failure str)
```

where `Failure` has type `Exception`. A caller of `failwith` always violates the contract `Any` because $\forall e, e \triangleleft$ `Any` $=$ `BAD`. For example:

```
let get a i = if i >= 0 and i < Array.length a - 1
          then a.(i) else failwith "Out of bound"
```

Whenever the else-branch is reached (either in SCC or DCC), the caller `get` is blamed because a safe program is meant not to *invoke* a function that fails. It is not useful to blame the `failwith` itself. Certainly, programmers' intention is not to have an index out of bound so they may give `get` a contract:

$$\{a \mid \texttt{true}\} \to \{i \mid i \geq 0 \wedge i < \texttt{Array.length}\ a - 1\} \to \{z \mid \texttt{true}\}$$

so that a caller of `get` will be blamed if it fails `get`'s precondition.

The example under debate in [12] is something like:

```
contract id = ({x | x /= 0} -> {y | true}) -> Any
let id x = x
let t3 = let invert y = 1/y in (id invert) 0
```

If programmers' intention is not to define a function that always fails, they should replace `Any` by `{z | true}`, which never assigns blame because $\forall e, e \triangleright \{z \mid$ `true`$\} = e \triangleleft \{z \mid$ `true`$\} = e$. With this new contract, `id` is blamed in either SCC or DCC for violating its contract because `id` cannot guarantee a crash-free

result (required by `{z | true}`) when taking a non-crash-free function as its argument.

With the declarative semantics for contract satisfaction, contracts can be exported for separate compilation. An implementation of a function may change over time (e.g. having a more efficient implementation), but its exported contract may not change. In our framework, we respect a function's contract more than its implementation. This is different from the original purpose in [13], which only uses contracts for dynamic blaming.

We have a simple lemma for contract `Any`.

Lemma 8 (Contract `Any`). *(a) If* $\texttt{BAD} \in t$*, then* $t = \texttt{Any}$.

(b) If $\texttt{BAD} \triangleright t$ *is crash-free, then* $t = \texttt{Any}$.

Proof. (a) By inspecting the definition of $\in$, the only contract that `BAD` satisfies is `Any`.

(b) By inspecting the definition of $\triangleright$, for all t such that $t \neq \texttt{Any}$, $\texttt{BAD} \triangleright t \rightarrow^* \texttt{BAD}$ which is not crash-free. And we have $\texttt{BAD} \triangleright \texttt{Any} = \texttt{UNR}$ which is crash-free, so we are done.

□

4.7 Contract ordering

the subcontract relation can be illustrated in rule-form shown in Figure 7. Each rule in Figure 7 is a theorem. The relation $p \Rightarrow_e q$ in rule [C-Pred] is defined in Definition 11. Rule [C-Any] follows directly from the definition of $\leqq$. We now study the rules [C-Pred], [C-DepFun] and [C-DepTup]. We assume the statement above the line is true, and prove the statement below the line is true. We leave the proof of other direction as a open problem.

$$\frac{p \Rightarrow_e q}{\{x \mid p\} \leqq \{x \mid q\}} \quad \text{[C-Pred]} \qquad t \leqq \texttt{Any} \quad \text{[C-Any]}$$

$$\frac{t_1 \leqq t_3 \qquad \forall e \in t_1, t_2[e/x] \leqq t_4[e/x]}{(x\colon t_1, t_2) \leqq (x\colon t_3, t_4)} \quad \text{[C-DepTup]}$$

$$\frac{t_3 \leqq t_1 \qquad \forall e \in t_3,\ t_2[e/x] \leqq t_4[e/x]}{x\colon t_1 \rightarrow t_2 \leqq x\colon t_3 \rightarrow t_4} \quad \text{[C-DepFun]}$$

Figure 7: Subcontract Relation

Definition 11 (Boolean Expression Implication). *For all boolean expressions* p *and* q*, we say* p *implies* q *(written* $p \Rightarrow_e q$*) iff* $\begin{pmatrix} \texttt{if } q \texttt{ then } () \\ \texttt{else BAD} \end{pmatrix} \preceq \begin{pmatrix} \texttt{if } p \texttt{ then } () \\ \texttt{else BAD} \end{pmatrix}$

From Definition 11[p21], for example, we know $\{x \mid x < 10\} \Rightarrow_e \{x \mid x < 12\}$.

The substitution for contracts is defined in Figure 8. Here, we assume each bound variable has a unique name.

$$
\begin{array}{lcl}
\{x \mid p\}[e/y] & = & \{x \mid p[e/y]\} \\
(x\colon t_1 \to t_2)[e/y] & = & x\colon t_1[e/y] \to t_2[e/y] \\
(t_1, t_2)[e/y] & = & (t_1[e/y], e_2[e/y]) \\
\texttt{Any}[e/y] & = & \texttt{Any}
\end{array}
$$

Figure 8: Substitution for Contracts

4.7.1 Predicate Contract Ordering

We prove that the rule [C-Pred] is sound; that is we prove Theorem 4[p22].

Theorem 4 (Predicate Contract Ordering)**.** *For all expressions p, q, if $p \Rightarrow q$ then $\{x \mid p\} \leqq \{x \mid q\}$.*

Proof. We have the following proof for all t_1, t_2, t_3, t_4:

$p \Rightarrow_e q$

$\iff$ (By Definition 11[p21] (Boolean Expression Implication), let
$e_1 = \begin{pmatrix} \texttt{case}\ p\ \texttt{of} \\ \texttt{True} \to () \\ \texttt{False} \to \texttt{BAD} \end{pmatrix}$ and $e_2 = \begin{pmatrix} \texttt{case}\ q\ \texttt{of} \\ \texttt{True} \to () \\ \texttt{False} \to \texttt{BAD} \end{pmatrix}$)
$e_2 \preceq e_1$

$\iff$ (By Definition 7[p14] (Crashes-more-often))
$\forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$

$\Rightarrow$ (By (*) below)
$\forall e.\ e$ is crash-free and $(e_1[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\} \Rightarrow e_2[e/x] \not\to^* \{\texttt{BAD}, \texttt{False}\})$

$\iff$ (By logic and definition of $\in$ in Figure 5)
$\forall e.\ e \in \{x \mid e_1\} \Rightarrow e \in \{x \mid e_2\}$

$\iff$ (By Definition 9[p17] (Subcontract))
$\{x \mid e_1\} \leqq \{x \mid e_2\}$

(*) We know $\forall e, a, x.\ e[a/x] \equiv_s \texttt{let}\ x = a\ \texttt{in}\ e$.
Assuming for all crash-free e:
(1) $\forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \to^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$
(2) $(\texttt{let}\ x = e\ \texttt{in}\ e_1) \not\to^* \{\texttt{BAD}, \texttt{False}\})$
we want to show $(\texttt{let}\ x = e\ \texttt{in}\ e_2) \not\to^* \{\texttt{BAD}, \texttt{False}\}$

Suppose $(\texttt{let}\ x = e\ \texttt{in}\ e_2) \to^* \texttt{BAD}$
By (1), let $\mathcal{C}$ be $\texttt{let}\ x = e\ \texttt{in}\ \bullet$, we have $\mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$.
That means $(\texttt{let}\ x = e\ \texttt{in}\ e_1) \to^* \texttt{BAD}$.
This contradicts with (2) so our assumption is wrong and we are done.

Suppose $(\texttt{let}\ x = e\ \texttt{in}\ e_2) \to^* \texttt{False}$
By (1), let $\mathcal{C}$ be $\texttt{case}\ (\texttt{let}\ x = e\ \texttt{in}\ \bullet)\ \texttt{of}\ \{\texttt{False} \to \texttt{BAD}\}$, we have $\mathcal{C}[\![e_1]\!] \to^* \texttt{BAD}$.
That means $(\texttt{case}\ (\texttt{let}\ x = e\ \texttt{in}\ e_1)\ \texttt{of}\ \{\texttt{False} \to \texttt{BAD}\}) \to^* \texttt{BAD}$.
That means $(\texttt{let}\ x = e\ \texttt{in}\ e_1) \to^* \{\texttt{BAD}, \texttt{False}\}$.
This contradicts with (2) so our assumption is wrong and we are done.

End of proof. □

4.7.2 Dependent Function Contract Ordering

We prove that the rule [C-DepFun] is sound; that is we prove Theorem 5[p23].

Theorem 5 (Dependent Function Contract Ordering)**.** *For all* t_1, t_2, t_3, t_4.

$$\textit{if } t_3 \leqq t_1 \textit{ and } \forall e \in t_3.\ t_2[e/x] \leqq t_4[e/x], \textit{ then } x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4$$

Proof. We have the following proof for all t_1, t_2, t_3, t_4:

$t_3 \leqq t_1$ and $\forall e_3 \in t_3.\ t_2[e_3/x] \leqq t_4[e_3/x]$
$\iff$ (By Definition 9[p17] (Subcontract))
($\dagger_1$) $\forall e_1.\ e_1 \in t_3 \Rightarrow e_1 \in t_1$ and $\forall e_3 \in t_3. \forall e_2.\ e_2 \in t_2[e_3/x] \Rightarrow e_2 \in t_4[e_3/x]$
$\Rightarrow$ (By the (*) below)
($\dagger_2$) $\forall e.\ \forall e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \Rightarrow \forall e_3 \in t_3.\ (e\ e_3) \in t_4[e_3/x]$
$\iff$ (By definition of $\in$ in Figure 5)
$\forall e.\ e \in x\colon t_1 \to t_2 \Rightarrow e \in x\colon t_3 \to t_4$
$\iff$ (By Definition 9[p17] (Subcontract))
$x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4$

($*$) For all e, assuming:
(1) $\forall e_1.\ e_1 \in t_3 \Rightarrow e_1 \in t_1$ (first clause of the line $\dagger_1$)
(2) $\forall e_3 \in t_3,\ \forall e_2.\ e_2 \in t_2[e_3/x] \Rightarrow e_2 \in t_4[e_3/x]$ (second clause of the line $\dagger_1$)
(3) $\forall e_1 \in t_1. (e\ e_1) \in t_2[e_1/x]$ (LHS of the line $\dagger_2$)
we show $\forall e_3.\ e_3 \in t_3 \Rightarrow (e\ e_3) \in t_4[e_3/x]$as follows.
$e_3 \in t_3$
$\iff$ (By (1))
$e_3 \in t_1$
$\iff$ (By (3))
$(e\ e_3) \in t_2[e_3/x]$
$\iff$ (By (2))
$(e\ e_3) \in t_4[e_3/x]$

We are done. □

4.7.3 Dependent tuple contract ordering

We prove the rule [C-DepTup] is sound by showing:

$$\text{For all } t_1, t_2, t_3, t_4.\ \text{if } t_1 \leqq t_3 \text{ and } t_2 \leqq t_4, \text{ then } (t_1, t_2) \leqq (t_3, t_4)$$

Proof. For all e, if e diverges, then for all t_1, t_2, t_3, t_4, $e \in (t_1, t_2)$ and $e \in (t_3, t_4)$ because a divergent expression satisfies all contracts. By the definition of $\leqq$, we have the desired result $(t_1, t_2) \leqq (t_3, t_4)$. Now, we prove the case when

$e \rightarrow^* (e_1, e_2)$ as follows.

$t_1 \leqq t_3$ and $t_2 \leqq t_4$
$\iff$ (By Definition 9[p17] (Subcontract))
$\forall e_1.\ e_1 \in t_1 \Rightarrow e_1 \in t_3$ and $\forall e_2.\ e_2 \in t_2 \Rightarrow e_2 \in t_4$
$\iff$ (By logic $(\forall x.A) \wedge (\forall y.B) \equiv \forall x, y.\ A \wedge B$ if $y \notin fv(A)$ and $x \notin fv(B)$)
$\forall e_1, e_2.\ e_1 \in t_1 \Rightarrow e_1 \in t_3$ and $e_2 \in t_2 \Rightarrow e_2 \in t_4$
$\Rightarrow$ (By logic $((A \Rightarrow B) \wedge (C \Rightarrow D)) \Rightarrow ((A \wedge C) \Rightarrow (B \wedge D)))$
$\forall e.\ e \rightarrow^* (e_1, e_2)$ and $((e_1 \in t_1$ and $e_2 \in t_2) \Rightarrow (e_1 \in t_3$ and $e_2 \in t_4))$
$\Rightarrow$ (By logic $(A \wedge (B \Rightarrow C)) \Rightarrow ((A \wedge B) \Rightarrow (A \wedge C)))$
$\forall e.\ (e \rightarrow^* (e_1, e_2)$ and $e_1 \in t_1$ and $e_2 \in t_2)$
$\Rightarrow (e \rightarrow^* (e_1, e_2)$ and $e_1 \in t_3$ and $e_2 \in t_4)$
$\iff$ (By definition of $\in$ in Figure 5)
$\forall e.\ e \in (t_1, t_2) \Rightarrow e \in (t_3, t_4)$
$\iff$ (By Definition 9[p17] (Subcontract))
$(t_1, t_2) \leqq (t_3, t_4)$

□

Note that some tuple contracts are not comparable by $\leqq$, for example: $(\texttt{Ok}, \texttt{Any}) \not\leqq (\texttt{Any}, \texttt{Ok})$ and $(\texttt{Any}, \texttt{Ok}) \not\leqq (\texttt{Ok}, \texttt{Any})$.

4.8 Contract equivalence

In this section we give formal definition of the equivalence of two contracts.

Definition 12 (Contract Equivalence)**.** *Two closed contracts t_1 and t_2 are equivalent, namely $t_1 \equiv_t t_2$, iff*

$$\forall e.\ e \in t_1 \iff e \in t_2$$

Contract equivalence $\equiv_t$ refers to semantic equivalence, not equality. For example, $\{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{true}\} \leqq \{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{false}\}$ and $\{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{false}\} \leqq \{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{true}\}$, and $\{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{true}\} \equiv_t \{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{false}\}$, but $\{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{true}\} \neq \{x \mid \texttt{false}\} \rightarrow \{x \mid \texttt{false}\}$.

Theorem 6 (Subcontract is antisymmetric)**.** *For all closed contracts t_1 and t_2, $t_1 \leqq t_2$ and $t_2 \leqq t_1$ iff $t_1 \equiv_t t_2$.*

Proof.

$t_1 \leqq t_2$ and $t_2 \leqq t_1$
$\iff$ (By Definition 9[p17] (Subcontract))
$\forall e.\ e \in t_1 \Rightarrow e \in t_2$ and $\forall e.\ e \in t_2 \Rightarrow e \in t_1$
$\iff$ (By logic $(\forall x.\ A(x) \Rightarrow B(x)) \wedge (\forall x.\ B(x) \Rightarrow A(x)) \equiv \forall x.\ A(x) \iff B(x)$)
$\forall e.\ e \in t_1 \iff e \in t_2$
$\iff$ (By Definition 12[p24] (Contract Equivalence))
$t_1 \equiv_t t_2$

End of proof. □

For open contracts t, we assume implicitly that there is an environment Δ, which is a mapping from variable to its type, contract and definition (See Definition 10[p19] in Section 4.4).

Lemma 9 (Predicate Contract Equivalence)**.** *For all expressions e_1 and e_2, if $e_1 \equiv_s e_2$, then $\{x \mid e_1\} \equiv_t \{x \mid e_2\}$.*

Proof. We have the following proof:

$$
\begin{array}{ll}
 & e_1 \equiv_s e_2 \\
\iff & \text{(By Theorem 1[p15] (Crashes-more-often is antisymmetric))} \\
 & e_1 \preceq e_2 \text{ and } e_2 \preceq e_1 \\
\iff & \text{(By Theorem 4[p22] (Predicate contract ordering))} \\
 & \{x \mid e_1\} \leqq \{x \mid e_2\} \text{ and } \{x \mid e_2\} \leqq \{x \mid e_1\} \\
\iff & \text{(By Theorem 6[p24] (Subcontract is antisymmetric))} \\
 & \{x \mid e_1\} \equiv_t \{x \mid e_2\}
\end{array}
$$

□

Lemma 10 (Dependent Function Contract Equivalence)**.** *For all contracts t_1, t_2, t_3, t_4, if $t_1 \equiv_t t_3$ and $\forall e \in t_1.\ t_2[e/x] \equiv_t t_4[e/x]$, then $x\colon t_1 \to t_2 \equiv_t x\colon t_3 \to t_4$.*

Proof. We have the following proof.

$$
\begin{array}{ll}
 & t_1 \equiv_t t_3 \text{ and } \forall e \in t_1.\ t_2[e/x] \equiv_t t_4[e/x] \\
\iff & \text{(By Theorem 6[p24] (Subcontract is Antisymmetric))} \\
 & t_1 \leqq t_3 \text{ and } t_3 \leqq t_1 \text{ and} \\
 & (\forall e \in t_1.\ t_2[e/x] \leqq t_4[e/x] \text{ and } \forall e \in t_1.\ t_4[e/x] \leqq t_2[e/x]) \\
\iff & \text{(Since } t_1 \equiv_t t_3, e \in t_1 \iff e \in t_3.\text{)} \\
 & t_3 \leqq t_1 \text{ and } \forall e \in t_1.\ t_2[e/x] \leqq t_4[e/x] \text{ and} \\
 & t_1 \leqq t_3 \text{ and } \forall e \in t_1.\ t_4[e/x] \leqq t_2[e/x] \\
\Rightarrow & \text{(By [C-DepFun] in Figure 7)} \\
 & x\colon t_1 \to t_2 \leqq x\colon t_3 \to t_4 \text{ and } x\colon t_3 \to t_4 \leqq x\colon t_1 \to t_2 \\
\iff & \text{(By Theorem 6[p24] (Subcontract is Antisymmetric))} \\
 & x\colon t_1 \to t_2 \equiv_t x\colon t_3 \to t_4
\end{array}
$$

We are done. □

Lemma 11 (Dependent Tuple Contract Equivalence)**.** *For all contracts t_1, t_2, t_3, t_4, if $t_1 \equiv_t t_3$ and $\forall e \in t_1.\ t_2[e/x] \equiv_t t_4[e/x]$, then $(x\colon t_1, t_2) \equiv_t (x\colon t_3, t_4)$.*

Proof. We have the following proof.

$$
\begin{array}{ll}
 & t_1 \equiv_t t_3 \text{ and } \forall e \in t_1.\ t_2[e/x] \equiv_t t_4[e/x] \\
\iff & \text{(By Theorem } 6^{\text{p24}} \text{ (Subcontract is Antisymmetric))} \\
 & t_1 \leqq t_3 \text{ and } t_3 \leqq t_1 \text{ and} \\
 & (\forall e \in t_1.\ t_2[e/x] \leqq t_4[e/x] \text{ and } \forall e \in t_1.\ t_4[e/x] \leqq t_2[e/x]) \\
\iff & \text{(Since } t_1 \equiv_t t_3, e \in t_1 \iff e \in t_3.) \\
 & t_3 \leqq t_1 \text{ and } \forall e \in t_1.\ t_2[e/x] \leqq t_4[e/x] \text{ and} \\
 & t_1 \leqq t_3 \text{ and } \forall e \in t_1.\ t_4[e/x] \leqq t_2[e/x] \\
\Rightarrow & \text{(By [C-DepFun] in Figure 7)} \\
 & (x\colon t_1, t_2) \leqq (x\colon t_3, t_4) \text{ and } (x\colon t_3, t_4) \leqq (x\colon t_1, t_2) \\
\iff & \text{(By Theorem } 6^{\text{p24}} \text{ (Subcontract is Antisymmetric))} \\
 & (x\colon t_1, t_2) \equiv_t (x\colon t_3, t_4)
\end{array}
$$

We are done. □

Theorem 7 (Subcontract and Crashes-more-often Ordering)**.** *For all* t_1 *and* t_2*,*

$$\forall e.\ e \triangleright t_1 \preceq e \triangleright t_2 \quad \Rightarrow \quad t_1 \leqq t_2$$

Proof. We have the following proof:

$$
\begin{array}{ll}
 & \forall e.\ e \triangleright t_1 \preceq e \triangleright t_2 \\
\Rightarrow & \text{(By Lemma } 7^{\text{p15}} \text{ (c) (Properties of Crashes-more-often - II))} \\
 & \forall e.\ e \triangleright t_1 \text{ is crash-free} \Rightarrow e \triangleright t_2 \text{ is crash-free} \\
\Rightarrow & \text{(By Theorem } 2^{\text{p18}} \text{ (grand theorem))} \\
 & \forall e.\ e \in t_1 \Rightarrow e \in t_2 \\
\iff & \text{(By Definition } 9^{\text{p17}} \text{ (Subcontract))} \\
 & t_1 \leqq t_2
\end{array}
$$

□

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 3, 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 for occurrence of `BAD` in e'.

We introduce an SL machine (Figure 10) which combines symbolic *simplification* and contextual information (ctx-info) synthesis 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;

$$\langle \mathcal{H} \mid n \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid n \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \quad \text{[S-const]}$$

$$\langle \mathcal{H} \mid r \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid r \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \quad \text{[S-exn]}$$

$$\langle \mathcal{H}[x \mapsto tval] \mid x \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle\!\langle \mathcal{H}[x \mapsto tval] \mid tval \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \quad \text{[S-var1]}$$

$$\text{if } x \notin \mathcal{H}, \qquad \langle \mathcal{H} \mid x \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid x \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \quad \text{[S-var2]}$$

$$\langle \mathcal{H} \mid \lambda x^{\tau}.e \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid e \mid (\lambda x.\bullet) :: \mathcal{S} \mid \mathcal{L}, \forall x : [\![\tau]\!] \rangle \quad \text{[S-lam]}$$

$$\langle \mathcal{H} \mid e_1\ e_2 \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid e_1 \mid (\bullet\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-app]}$$

$$\langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ alts \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid e_0 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-match]}$$

$$\langle \mathcal{H} \mid K\ (a_1, \ldots, e_i, \ldots, e_n) \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid e_i \mid (K\ (a_1, \ldots, \bullet, \ldots, e_n)]) :: \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-K]}$$

$$\text{if } x \notin fv(e), \quad \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid (\bullet\ e) :: \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ e \mid \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-letL]}$$

$$\text{if } fv(e) \cap \overrightarrow{x_i} = \emptyset, \quad \langle \mathcal{H} \mid (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i}) \mid (\bullet\ e) :: \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i\ e} \mid \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-matchL]}$$

$$\text{if } x \notin fv(a), \quad \langle \mathcal{H} \mid val \mid (\bullet\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)) :: \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ val\ e_2 \mid \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-letR]}$$

$$\text{if } fv(val) \cap \overrightarrow{x} = \emptyset, \quad \langle \mathcal{H} \mid val \mid (\bullet\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})) :: \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to val\ e} \mid \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-matchR]}$$

$$\text{if } fv(alts) \cap \overrightarrow{x} = \emptyset, \quad \langle \mathcal{H} \mid \overrightarrow{\begin{array}{l}\texttt{match}\ e_0\ \texttt{with}\\ K\ \overrightarrow{x} \to e\end{array}} \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid \overrightarrow{\begin{array}{l}\texttt{match}\ e_0\ \texttt{with}\\ K\ \overrightarrow{x} \to \texttt{match}\ e\ \texttt{with}\ alts\end{array}} \mid \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-match-match]}$$

$$\text{if } x \notin fv(alts), \quad \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H} \mid \texttt{let}\ x = e_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts \mid \mathcal{S} \mid \mathcal{L} \rangle \quad \text{[S-match-let]}$$

Figure 9: SL machine part (a)

$\langle\!\langle \mathcal{H} \mid a \mid [\,] \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow a$ [R-done]

if $(s \neq \texttt{match}\ e\ \texttt{with}\ K\ \overrightarrow{x} \rightarrow (\bullet, \mathcal{S}, \mathcal{L}))$,
$\langle\!\langle \mathcal{H} \mid r \mid s :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid r \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ [R-r]

$\langle\!\langle \mathcal{H} \mid a \mid (\lambda x.\bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid \lambda x.a \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ [R-lam]

Rules below: $a \notin \{\texttt{BAD}^l, \texttt{UNR}^l\}$

$\langle\!\langle \mathcal{H} \mid a \mid (\bullet\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle \mathcal{H} \mid e_2 \mid (a\ \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle$ [R-fun]

$\langle\!\langle \mathcal{H} \mid val \mid ((\lambda x.a_1)\ \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{H}[x \mapsto val] \mid a_1 \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ [R-beta]

if $a_1 \neq \lambda x.a'$ or $a \neq val$,
$\langle\!\langle \mathcal{H} \mid a \mid (a_1\ \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid a_1\ a \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ [R-app]

$\langle\!\langle \mathcal{H} \mid a_n \mid (K\ a_1 \ldots \bullet) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{H} \mid K\ \overrightarrow{a} \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ [R-K]

$\langle\!\langle \mathcal{H} \mid K\ \overrightarrow{a} \mid (\texttt{match}\ \bullet\ \texttt{with}\ \{\ldots; K\ \overrightarrow{x} \rightarrow e; \ldots\}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$
$\rightsquigarrow \langle \mathcal{H} \mid \texttt{let}\ \overrightarrow{x = a}\ \texttt{in}\ e \mid \mathcal{S} \mid \mathcal{L} \rangle$ [R-K-match]

if exists $(K\ \overrightarrow{x^\tau})$ such that $\mathcal{L} \Rightarrow (\exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})})$,
$\langle\!\langle \mathcal{H} \mid a \mid (\texttt{match}\ \bullet\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^\tau} \rightarrow e}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$
$\rightsquigarrow \langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L}, \exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})} \rangle$ [R-s-match]

if for all $(K\ \overrightarrow{x^\tau})$ such that $\mathcal{L} \not\Rightarrow (\exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})})$,
$\langle\!\langle \mathcal{H} \mid a \mid (\texttt{match}\ \bullet\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^\tau} \rightarrow e}) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$
$\rightsquigarrow \langle \mathcal{H} \mid e \mid \begin{array}{l}(\texttt{match}\ a\ \texttt{with}\ K\ \overrightarrow{x^\tau} \\ \rightarrow (\bullet, \mathcal{S}, \mathcal{L})) :: [\,]\end{array} \mid \begin{array}{l}\mathcal{L}, \exists \overrightarrow{x : [\![\tau]\!]}, \\ [\![a]\!]_{(K\ \overrightarrow{x})}\end{array} \rangle$ [R-s-save]

$\langle\!\langle \mathcal{H} \mid a \mid (\texttt{match}\ a_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow (\bullet, \mathcal{S}, \mathcal{L})}) :: \mathcal{S}' \mid \mathcal{L}' \rangle\!\rangle$
$\rightsquigarrow \langle\!\langle \mathcal{H} \mid \texttt{match}\ a_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \rightarrow a} \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$ for some $\mathcal{S}'$ and $\mathcal{L}'$ [R-match]

$\langle\!\langle \mathcal{H} \mid a \mid (\texttt{let}\ x^\tau = \bullet\ \texttt{in}\ e_2) :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle$
$\rightsquigarrow \langle \mathcal{H} \mid e_2 \mid (\texttt{let}\ x = a\ \texttt{in}\ \bullet) :: \mathcal{S} \mid \mathcal{L}, \exists x\colon [\![\tau]\!], [\![a]\!]_x \rangle$ [R-let-save]

Figure 10: SL machine part (b)

- 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;

$$(\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ e \Longrightarrow \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ e \qquad \text{[letL]}$$

$$\text{if } fv(e) \cap \overrightarrow{x} = \emptyset,$$
$$(\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i})\ e$$
$$\Longrightarrow \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to (e_i\ e)} \qquad \text{[matchL]}$$

$$\text{if } x \notin fv(e),$$
$$tval\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \Longrightarrow \texttt{let}\ x = e_1\ \texttt{in}\ tval\ e_2 \qquad \text{[letR]}$$

$$\text{if } fv(tval) \notin \overrightarrow{x},$$
$$val\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})$$
$$\Longrightarrow \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to val\ e} \qquad \text{[matchR]}$$

$$\text{if } fv(alts) \cap \overrightarrow{x} = \emptyset,$$
$$\texttt{match}\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})\ \texttt{with}\ alts$$
$$\Longrightarrow \texttt{match}\ e_o\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to \texttt{match}\ e\ \texttt{with}\ alts} \qquad \text{[match-match]}$$

$$\text{if } x \notin fv(alts),$$
$$\texttt{match}\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ \texttt{with}\ alts$$
$$\Longrightarrow \texttt{let}\ x = e_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts \qquad \text{[match-let]}$$

$$\texttt{match}\ K\ a_1 \ldots a_n\ \texttt{with}\ \{\ldots; K\ x_1 \ldots x_n \to e; \ldots\}$$
$$\Longrightarrow \texttt{let}\ x_1 = a_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = a_n\ \texttt{in}\ e \qquad \text{[scrut-match]}$$

Figure 11: Simplification Rules

- $\mathcal{L}$ 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 12.

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

5.1 The SL machine

In Figure 10, 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 [16]), 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

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

The transitions [S-app], [S-match] and [S-K] *implement* the context reduction in Figure 3. The transitions [S-letL], [S-matchL], [S-letR], [S-matchR], [S-match-match], [S-match-let] *implement* the conventional simplification rules in Figure 11. Here, $\overrightarrow{x}$ 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

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

(where e is a crash-free expression, not a value) to $\texttt{let}\ x = e\ \texttt{in}\ 5$ 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 transitions [R-r-match], [R-r-let], [R-r-fun] and [R-r-arg] rebuild `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 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:

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

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

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

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

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

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 $\exists(K\ \overrightarrow{x})$, $\mathcal{L} \Rightarrow [\![a]\!]_{(K\ \overrightarrow{x})}$" checks if there is any branch $K\ \overrightarrow{x}$ that matches the scrutinee. 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 $[\![a]\!]_{(K\ \overrightarrow{x})}$ (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.

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

In the `true` branch, after a few steps, we rebuild the scrutinee $x + 1 > 0$. In this case, $\forall x : \texttt{int}, x > 0 \Rightarrow x + 1 > 0$ 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:

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

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

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

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

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

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

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

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

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

After a few simplification steps, we have:

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

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.

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

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

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

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

Theorem 8 (SL machine terminates)**.** *For all expression e, there exists an expression a such that $\langle \emptyset \mid e \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* a$.*

Proof. See Appendix B.2. □

Intuitively, SL machine behaves like CEK machine [16], but *does not inline top-level functions* and we do not have local `let rec` in our language. We also call SMT solver Alt-ergo with an option "-stop <time-bound>" or "-steps <bound>" to make sure the SMT solver terminates. So there is no element causing non-termination.

Theorem 9 (Correctness of SL machine)**.** *For all expression e, if $\langle \emptyset \mid e \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* a$, then $e \equiv_s a$.*

Proof. See Appendix B.2. □

The SL is designed in a way such that the simplified a preserves the semantics of the original expression e. The proof of Therem 9 (in Appendix B.2) 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 $e_1 \equiv_s e_2$.

Theorem 10 (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 9, Lemma 1 and Theorem 3. □

5.2 Logicization

We now explain the mysterious convertion $[\![.]\!]_f$, which we call *logicization*. Figure 12 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, data type declaration in language M, e.g.

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

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

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

$$
\begin{array}{rcll}
x, s, i, f & \in & \textbf{Identifier} & \\
file & ::= & decl_1, \ldots, decl_n & \\
bty & ::= & \texttt{int} \mid \texttt{bool} \mid i \mid \texttt{'}i \mid \overrightarrow{bty}\ i & \textbf{Base type} \\
lty & ::= & bty \mid \vec{ty}\ \texttt{->}\ bty & \textbf{Logic type} \\
ty & ::= & \alpha \mid (ty_1, \ldots, ty_n)\ s & \textbf{Types} \\
decl & ::= & \texttt{type}\ \vec{\texttt{'}i}\ s & \\
 & \mid & \texttt{logic}\ \vec{i}\ : lty \mid \texttt{axiom}\ i : \phi \mid \texttt{goal}\ i : \phi & \\
\oplus & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} & \\
\odot_t & ::= & \texttt{=} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} & \\
\odot_p & ::= & \texttt{->} \mid \texttt{<->} \mid \texttt{or} \mid \texttt{and} & \\
m & ::= & n \mid x \mid m_1 \oplus m_2 \mid \texttt{-}\ m \mid f\ \overrightarrow{m} & \textbf{Term} \\
\phi & ::= & true \mid false \mid f\ \overrightarrow{m} & \textbf{Predicate} \\
 & \mid & m_1\ \odot_t\ m_2 \mid \phi_1\ \odot_p\ \phi_2 \mid not(\phi) & \\
 & \mid & \texttt{forall}\ \vec{x} : ty.\phi \mid \texttt{exists}\ \vec{x} : ty.\phi &
\end{array}
$$

Figure 12: Syntax of logic declaration

Data type in language M:

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

Corresponding alt-ergo code:

$\texttt{type}\ \overrightarrow{\texttt{'}a}\ s$

$\texttt{logic}\ K_1 : \overrightarrow{t_1}\ \texttt{->}\ \overrightarrow{\texttt{'}a}\ s$

:

$\texttt{logic}\ K_n : \overrightarrow{t_n}\ \texttt{->}\ \overrightarrow{\texttt{'}a}\ s$

Figure 13: Converting data type to Alt-ergo code

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 convertion algorithm for an arbitrary user-defined data type is in Figure 13.

Moreover, we introduce a first order function type:

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

which allows us to encode the function type in the langugage 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 [22]. Converting types in the language M is straight forward (Figure 14).

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

Figure 14: Converting higher order type to first order type

We now give an example showing that the SL machine is better than the unrolling approach in [37, 40][1].

```
(* 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 $e_{\mathtt{a}}$ and $t_{\mathtt{a}}$ stand for the definition and contract of `append` respectively. Applying only simplification rules (including reduction rules) to $e_{\mathtt{a}} \triangleright t_{\mathtt{a}}$, we get (R3):

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

The simplification approach in [37] and the model checking approach in [32] 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, 9, 27, 6], 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 $[\![\mathtt{BAD}]\!]$ and $[\![\mathtt{UNR}]\!]$? Our key idea is not to convert an expression directly to a corresponding logical term, but form equality with $[\![.]\!]_f$ recursively (defined in Figure 15). 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

[1] Unrolling approach may suit a lazy language better.

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 [9, 33, 27, 6]) deal with non-total terms. The work [6] uses approaches in [9, 27] 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, in the goal `app_1`, the ctx-info `v1=nil` is from the pattern matching `match v1 with {[] -> ....}`; the query is the scrutinee `apply(len,v2) = apply(len,v1) + apply(len,v2)`. Alt-ergo says *valid* for both goals.

First, how to systematically convert a function definition to an axiom (e.g. `len_def_1`)? Figure 15 gives an operator $[\![.]\!]_f$ that converts an expression to a logical formula. The subscript f in $[\![e]\!]_f$ denotes the expression e. For example, we can get `len_def_1` thus:

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

Let x_0 be s, we get a more readable version (axiom `len_def_1`).

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

Figure 15: Convert expression to logical formula

Theorem 11 (Logicization for axioms)**.** *Given definition $f = e^\tau$, $\forall fv(e), \exists f : \tau, [\![e]\!]_f$ is valid.*

Proof. See Appendix B.1. □

Next, what query (i.e. goal) shall we make? All we want is to check 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 $\mathcal{L} = \forall v_1 :$ `'a list`$, \forall v_2 :$ `'a list`$, v_1 =$ `nil` by [S-lam] and [R-s-save]. The scrutinnee is $[\![$`len` $v_2 =$ `len` $v_1 +$ `len` $v_2\}]\!]_{\texttt{true}}$. That is, we want to check whether `len` $v_2 =$ `len` $v_1 +$ `len` v_2 is equivalent to `true`. Sending the Alt-ergo code in this paper to Alt-ergo solver, it replies *valid* for both goals. Thus, we know both $\texttt{BAD}^{l1}$ and $\texttt{BAD}^{l2}$ are not reachable.

Theorem 12 (Logicization for goals: validity preservation)**.** *For all (possibly open) expression e^τ, $\exists f : \tau$, if $\forall fv(e) : \tau, [\![e]\!]_f$ is valid and $e \rightarrow e'$ for some e', then $\forall fv(e'), [\![e']\!]_f$ is valid.*

Proof. See Appendix B.1. □

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 15, there are two convertions 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 crash-free 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 Section 4.5) 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 $\forall x\colon \texttt{int}, \exists y\colon \texttt{int}, (x>0 \Rightarrow y=x) \wedge (\mathit{not}(x>0) \Rightarrow \mathit{false})$. So $\mathcal{L} \Rightarrow y+1>0$ is $\forall x\colon \texttt{int}, \exists y\colon \texttt{int}, (x>0 \Rightarrow y=x) \wedge (\mathit{not}(x>0) \Rightarrow \mathit{false}) \Rightarrow y+1>0$, which is valid. It means, if $\mathit{not}(x>0)$ 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 3. 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.

$$
\begin{array}{rcll}
\Delta(n) & = & n & \text{[D1]} \\
\Delta(x) & = & x \quad \text{if } x \notin \mathit{dom}(\Delta) \text{ or } [x \mapsto \bot] \subseteq \Delta & \text{[D2]} \\
\Delta[x \mapsto m](x) & = & m & \text{[D3]} \\
\Delta(\exists x : ty, x = m \wedge \phi_1) & = & \Delta[x \mapsto \Delta(m)](\phi_1) & \text{[D4]} \\
\Delta(m_1 \odot_t m_2) & = & \Delta(m_1) \odot_t \Delta(m_2) & \text{[D5]} \\
\Delta(\phi_1 \odot_p \phi_2) & = & \Delta(\phi_1) \odot_p \Delta(\phi_2) & \text{[D6]} \\
\Delta(\forall x : ty.\ \phi_1) & = & \forall x : ty,\ \Delta(\phi_1) & \text{[D7]}
\end{array}
$$

Figure 16: Partial elimination of $\exists$ quantifiers

One might notice that the logicization generates some existentially quantified variables and simple equalities which can be easily eliminated. By observing the conversion in Figure 15, we may encounter some sub-formula in this form: $\exists x : ty, x = m \wedge \phi$, which can be simplified to $\phi[m/x]$. A simple $\exists$-elimination algorithm in Figure 16 is good enough to eliminate some (but not all) existential quantifiers from the formula. The environment Δ captures the maping from an $\exists$-bound variable to a term. For example:

$$
\begin{array}{rl}
 & \Delta(\forall y : \texttt{int}, \exists x : \texttt{int}, x = y \wedge (\exists x : \texttt{int}, x = 8 \wedge x > 6)) \\
= & \text{(By [D7])} \\
 & \forall y : \texttt{int}, \Delta(\exists x : \texttt{int}, x = y \wedge (\exists x : \texttt{int}, x = 8 \wedge x > 6)) \\
= & \text{(By [D4])} \\
 & \forall y : \texttt{int}, \Delta[x \mapsto y](\exists x : \texttt{int}, x = 8 \wedge x > 6) \\
= & \text{(By [D4])} \\
 & \forall y : \texttt{int}, \Delta[x \mapsto 8](x > 6) \\
= & \text{(By [D5])} \\
 & \forall y : \texttt{int}, \Delta[x \mapsto 8](x) > \Delta[x \mapsto 8](6) \\
= & \text{(By [D1] and [D0])} \\
 & \forall y : \texttt{int}, 8 > 6
\end{array}
$$

The $\Delta[x \mapsto \Delta(m)]$ means that, if $x \notin dom(\Delta)$, we extend the environment Δ with $[x \mapsto \Delta(m)]$; if $x \in dom(\Delta)$, we update x with the term $\Delta(m)$. The rest is self-explanatory.

Theorem 13 (Correctness of $\exists$ quantifiers elimination)**.** *For all FOL formula ϕ, $\Delta(\phi)$ is valid if and only if ϕ is valid.*

Proof. The only change to the formula ϕ is to substitute the existentially quantified x by m. Since we have the equality $x = m$ and the conjunction, it is immediate that the substitution is correct. □

One might worry that the rule [match-match] causes exponential code explosion for static analysis (although no run-time overhead). For example, $h_1 =$ `if` (a `then` b `else` c) `then` d `else` e, where a, b, c, d, e are expressions. At program point d, the ctx-info is $(a \Rightarrow b) \wedge (not(a) \Rightarrow c)$[2]. Applying [match-match] to h_1, we get: $h_2 =$ `if` a `then` (`if` b `then` d `else` e) `else` (`if` c `then` d `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: `if` a `then` (`if` b `then` d `else` e) `else` c. So d is not duplicated and we have smaller formula for the SMT solver.

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

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

to `if` $x_1 > 0$ `then` $(\lambda v.v + 1)\ x_1$ `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: `if` $x_1 > 0$ `then` $x_1 + 1$ `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[3] based on the source code of ocamlc-3.11.2. Table 1 shows the results of preliminary experiments, which are done on a PC running Ubuntu Linux with quadcore 2.93GHz CPU and 3.2GB memory. We take some examples from [25] and OCaml stdlib and time the static checking. The column Ann gives the LOC for contract annotations.

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.

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

[3] http://gallium.inria.fr/~naxu/research/hcc.html

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

6 Hybrid contract checking

We have explained with examples how SCC, DCC, HCC work in Section 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 Section 2).

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

We have:

$$\begin{aligned}
&[\text{SCC}]: \ \texttt{sl}(e_i[(f_j \lhd^{f_j}_{f_i} t_{f_j})/f_j] \rhd^{f_i}_{?} t)\\
&[\text{DCC}]: \ e_i[(f_j \underset{\text{BAD}^{f_i}}{\overset{\text{BAD}^{f_j}}{\bowtie}} t_{f_j})/f_j]\\
&[\text{HCC}]: \ f_i\sharp = \lambda?.\texttt{rmUNR}(\texttt{sl}(e_i[((f_j\sharp\ \text{“}f_i\text{”}) \lhd^{f_j}_{f_i} t_{f_j})/f_j] \rhd^{f_i}_{?} t))
\end{aligned}$$

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 Section 5.2 (although it can), recalling R3 in Section 5.2, the residual code `append`$\sharp$ is:

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

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 12 (Telescoping property [7, 39])**.** *For all expression e, total contract t, blames r_1, r_2, r_3, r_4,* $\quad (e \underset{r_2}{\overset{r_1}{\bowtie}} t) \underset{r_4}{\overset{r_3}{\bowtie}} t = e \underset{r_4}{\overset{r_1}{\bowtie}} t$.

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 $(f_j \underset{\text{UNR}^{f_i}}{\overset{\text{BAD}^{f_j}}{\bowtie}} t_{f_j}) \underset{\text{BAD}^{f_i}}{\overset{\text{UNR}^{f_j}}{\bowtie}} t_{f_j}$. By the telescoping property, we have:

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

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 $\texttt{BAD}^l$ is invoked before $\texttt{UNR}^l$ for the same l. That is, (`if` p `then` e_1 `else` $\texttt{BAD}^l$) is invoked before (`if` p `then` e `else` $\texttt{UNR}^l$) 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 (`if` p `then` e_1 `else BAD`) diverges or crashes first. It is easy to see if $t = \{x \mid p\}$. If $t = t_1 \to t_2$, then $(e \overset{\texttt{BAD}^{f_j}}{\underset{\texttt{UNR}^{f_i}}{\bowtie}} t_1 \to t_2) \overset{\texttt{UNR}^{f_j}}{\underset{\texttt{BAD}^{f_i}}{\bowtie}} t_1 \to t_2$ expands to

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

Focusing on the `BAD`s and `UNR`s above $\bowtie$, inspecting [P1] and [P2] in Figure 6, we can see that $\texttt{BAD}^{f_j}$ is invoked before $\texttt{UNR}^{f_j}$ and $\texttt{BAD}^{f_i}$ is invoked before $\texttt{UNR}^{f_i}$.

7 Related work

Contract semantics were first formalized in [7, 12] for a strict language and later in [39] 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 [18, 15, 2, 33] 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, 40], recursive contracts [7], polymorphic contracts [3]. Contracts also enjoy interesting mathematical properties [7, 12, 39, 38]. We like the idea of ghost refinement in [35] 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, ?]. 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 [35] invents a *Kind* checker to report ill-formed refinement types. As discussed in §4.3, 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 [?] (for verifying C and Java code) suggests that symbolic execution is faster than verification condition generation method [15, 2].

The work [23] mixes type checking and symbolic execution. However, [23] 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 [23] 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 [35], 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 [31, 24, 32, 25] 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 [21] 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 [20, 19], which makes conversion of higher order functions easier. But we use the technique in different contexts.

Many papers on program verification [36, 15, 2, 30, 29, 11] 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 [?] 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, 29, 21] are inspiring.

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

References

[1] Nicolas Ayache and Jean-Christophe Filliatre. Combining the Coq proof assistant with first-order decision procedures. Unpublished, 2006.

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

[3] João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In Gilles Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2011.

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

[5] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33:8:1–8:45, February 2011.

[6] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with smt solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

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

[8] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[9] Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2007.

[10] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias 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. ACM.

[11] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *FoVeOOS*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010.

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

[13] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.

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

[15] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie 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, New York, NY, USA, 2002. ACM Press.

[16] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias 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.

[17] Michael Greenberg, Benjamin C. Pierce, and Stephanie 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.

[19] Kohei Honda, Martin Berger, and Nobuko Yoshida. Descriptive and relative completeness of logics for higher-order functions. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 360–371. Springer, 2006.

[20] Kohei Honda and Nobuko 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, New York, NY, USA, 2004. ACM Press.

[21] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. Hmc: Verifying functional programs using abstract interpreters. In *the 15th international conference on Computer Aided Verification CAV*, pages 262–274, 2011.

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

[23] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey 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.

[24] Naoki 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, New York, NY, USA, 2009. ACM.

[25] Naoki Kobayashi, Ryosuke Sato, and Hiroshi 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. ACM.

[26] Chin Soon Lee, Neil D. Jones, and Amir 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.

[27] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

[28] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[29] Matthew 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, New York, NY, USA, 2007. ACM.

[30] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In John H. Reppy and Julia L. Lawall, editors, *ICFP*, pages 62–73. ACM, 2006.

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

[32] C.-H. Luke Ong and Steven James 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, New York, NY, USA, 2011. ACM.

[33] Yann Régis-Gianas and François Pottier. A hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *MPC*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, 2008.

[34] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005.

[35] Nikhil Swamy, Juan Chen, Cedric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, 2011.

[36] Hongwei Xi and Frank 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, New York, NY, USA, 1999.

[37] Dana N. Xu. Extended static checking for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 48–59, New York, NY, USA, 2006.

[38] Dana N. Xu. Hybrid contract checking. INRIA research report, 2011.

[39] Dana N. Xu, Simon Peyton Jones, and Koen 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, New York, NY, USA, 2009. ACM.

[40] Na Xu. *Static Contract Checking for Haskell.* Ph.D. thesis, August 2008.

A Proof for the main theorem

The proof in this Section is similar to the one in [40] but for a call-by-value language.

$$
\begin{array}{lcl}
|.| & :: & \textbf{Contract} \to \textbf{Int} \\
|\{x \mid p\}| & = & 1 \\
|x\colon t_1 \to t_2| & = & |t_1| + |t_2| + 1 \\
|(t_1, t_2)| & = & |t_1| + |t_2| + 1 \\
|\texttt{Any}| & = & 1
\end{array}
$$

Figure 17: Size of Contract

As some of the proofs involve the structural induction on the size of contract, we define it in Figure 17. To make the proof look less clustered, we use the following shorthands:

cf	:	crash-free
ss	:	syntatically safe
defn	:	definition
cl	:	closed
tl	:	total

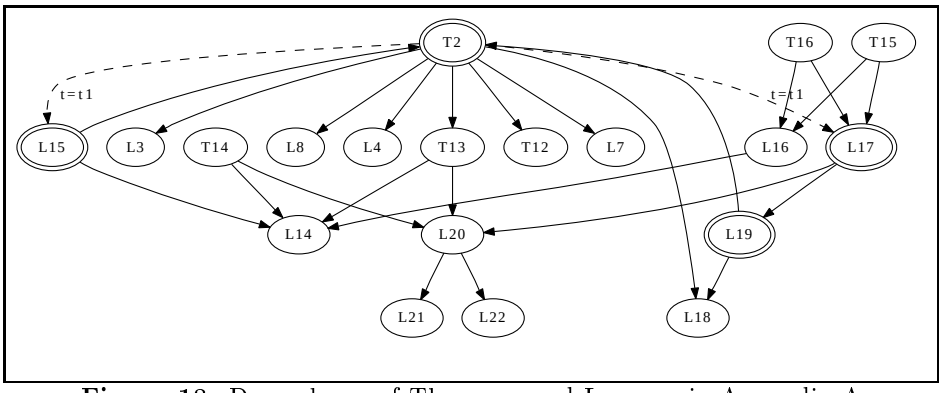

Figure 18: Dependency of Theorems and Lemmas in Appendix A

To make the dependency of theorems and lemmas clear, a dependency diagram is shown in Figure 18. For many theorems and lemmas, we prove them by induction on the size of contract t. The dashed directed edge shows that the size of the contract decreases, i.e. for a function contract $x\colon t_1 \to t_2$, we

call another lemma (or theorem) with $t = t_1$ or $t = t_2$. The solid directed edge shows the size of the contract is preserved. This makes the proof well-founded even though there are cycles in the dependencies (examined in Section A.3).

Theorem 2 (Soundness and Completeness of Contract Checking (grand theorem)) For all closed expression e^τ, closed and total contract t^τ,

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

There are two directions to be proved:

- $e \in t \Rightarrow e \triangleright t$ is crash-free. The difficulty lies in the proof for dependent function contracts. We appeal to a key lemma (Lemma 14^{p53} [Key lemma] in Section A.2).
- $e \triangleright t$ is crash-free $\Rightarrow e \in t$. The difficulty also lies in the proof for dependent function contracts. We appeal to three things:
  - definition and properties of crashes-more-often (Definition 7^{p14}, Lemma 7^{p15}).
  - projection pair property of $\triangleright$ and $\triangleleft$ (Theorem 15^{p55} in Section A.5);
  - congruence of crashes-more-often (Theorem 14^{p55} in Section A.4).

Proof. The notation e^τ and t^τ mean that both the expression e and the contract t are well-typed and they have the same type τ. The proof begins by dealing with two special cases:

- Case $e \rightarrow^* \texttt{BAD}$: We prove the two directions separately.

  ($\Rightarrow$)

  $e \triangleright t$ is **cf**
  $\Rightarrow$ (By Lemma 3^{p13} (preservation of crash-freeness) and Lemma 8^{p21}(b) (about `Any`))
  $t = \texttt{Any}$
  $\Rightarrow$ (By defn of $\in$, every expression satisfies `Any`)
  $e \in t$

  ($\Leftarrow$)

  $e \in t$
  $\Rightarrow$ (By Lemma 3^{p13} (preservation of crash-freeness) and Lemma 8^{p21}(a) (about `Any`))
  $t = \texttt{Any}$
  $\Rightarrow$ (By defn of $\triangleright$)
  $e \triangleright \texttt{Any}$ is crash-free

- Case $e \uparrow$: By inspecting the definition of $\triangleright$ and $\in$, for all t, if $e \uparrow$, then $(e \triangleright t) \uparrow$ and $e \in t$. Thus, we are done.

Hence, for the rest of the proof, we assume that $e \rightarrow^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$.
The rest of the proof is by induction on the size of t.

- Case t is $\{x \mid p\}$:

$$\begin{array}{ll} & e \triangleright \{x \mid p\} \text{ is } \mathbf{cf} \\ \iff & \text{(By defn of } \triangleright\text{)} \\ & \left(\begin{array}{l} \texttt{let } x = e \texttt{ in} \\ \texttt{match } p \texttt{ with} \\ \mid \texttt{true} \to x \\ \mid \texttt{false} \to \texttt{BAD} \end{array}\right) \text{ is } \mathbf{cf} \\ \iff & \text{(Since } e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}\text{)} \\ & e \text{ is } \mathbf{cf} \text{ and } p \not\to^* \{\texttt{BAD}, \texttt{false}\} \\ \iff & \text{(By defn of } \in\text{)} \\ & e \in \{x \mid p\} \end{array}$$

- Case t is $x\colon t_1 \to t_2$: we want to prove that

$$(e \triangleright x\colon t_1 \to t_2) \text{ is } \mathbf{cf} \iff e \in x\colon t_1 \to t_2$$

We have the following induction hypotheses:

$$\begin{array}{ll} \forall \mathbf{cl}\ e_1,\ e_1 \triangleright t_1 \text{ is } \mathbf{cf} \iff e_1 \in t_1 & \text{[IH1]} \\ \forall \mathbf{cl}\ e_2, \mathbf{cl\ tl}\ e'.\ e_2 \triangleright t_2[e'/x] \text{ is } \mathbf{cf} \iff e_2 \in t_2[e'/x] & \text{[IH2]} \end{array}$$

We have the following proof:

$$\begin{array}{ll} & e \triangleright x\colon t_1 \to t_2 \text{ is } \mathbf{cf}. \\ \iff & \text{(By defn of } \triangleright\text{)} \\ & \texttt{let } y = e \texttt{ in } \lambda x_1.(y\ (x_1 \triangleleft t_1)) \triangleright t_2[(x_1 \triangleleft t_1)/x] \text{ is } \mathbf{cf}. \\ \iff & \text{(Since } e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}\text{)} \\ & \lambda x_1.\ (e\ (x_1 \triangleleft t_1)) \triangleright t_2[(x_1 \triangleleft t_1)/x] \text{ is } \mathbf{cf}. \\ \iff & \text{(By Lemma } 4^{\text{p13}} \text{ (crash-free function))} \\ (\dagger) & \forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \triangleright t_2[(e' \triangleleft t_1)/x] \text{ is } \mathbf{cf}. \end{array}$$

Now the proof splits into two. In the reverse direction, we start with the assumption $e \in x\colon t_1 \to t_2$:

$$\begin{array}{ll} & e \in x\colon t_1 \to t_2 \\ \iff & \text{(By defn of } \in\text{)} \\ & \forall\, e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x] \\ \Rightarrow & \text{(By Lemma } 14^{\text{p53}} \text{ (Key lemma), let } e_1 = e' \triangleleft t_1\text{)} \\ & \forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \in t_2[(e' \triangleleft t_1)/x]) \\ \iff & \text{(By [IH2])} \\ (\dagger) & \forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \triangleright t_2[(e' \triangleleft t_1)/x] \text{ is } \mathbf{cf}. \end{array}$$

Now we have reached the desired conclusion (†). The key step is the use of Lemma 14^{p53} (Key lemma) (see Section A.2).

In the forward direction, we start with (†):

$$
\begin{array}{ll}
 & \forall \mathbf{cf}\ e'.\ (e\ (e' \triangleleft t_1)) \triangleright t_2[(e' \triangleleft t_1)/x] \text{ is } \mathbf{cf}. \\
\Rightarrow & \text{(By [IH1]}, e_1 \in t_1 \Rightarrow (e_1 \triangleright t_1) \text{ is } \mathbf{cf} \text{ so we replace } e' \text{ by } e_1 \triangleright t_1) \\
 & \forall e_1 \in t_1.\ (e\ ((e_1 \triangleright t_1) \triangleleft t_1)) \triangleright t_2[(e_1 \triangleright t_1 \triangleleft t_1)/x] \text{ is } \mathbf{cf} \\
\Rightarrow & \text{(By (Theorem } 15^{\text{p55}} \text{ (projection pair) and} \\
 & \text{Theorem } 14^{\text{p55}} \text{ (congruence of } \preceq\text{) and} \\
 & \text{Lemma } 7^{\text{p15}} \text{ (c) (property of } \preceq\text{)) twice)} \\
 & \forall e_1 \in t_1.\ (e\ e_1) \triangleright t_2[e_1/x]) \text{ is } \mathbf{cf} \\
\Rightarrow & \text{(By [IH2])} \\
 & \forall\, e_1 \in t_1.\ (e\ e_1) \in t_2[e_1/x]) \\
\Longleftrightarrow & \text{(by definition of } \in\text{)} \\
 & e \in x\colon t_1 \to t_2
\end{array}
$$

There are two key steps: one is to choose a *particular* crash-free e', namely $(e_1 \triangleright t_1)$ where $e_1 \in t_1$; the other one is the appeal to Theorem 15^{p55}, the projection pair property of $\triangleright$ and $\triangleleft$ (see Section A.5).

- t is $(x\colon t_1, t_2)$: We have the following induction hypotheses:

$$
\begin{array}{ll}
\forall \mathbf{cl}\ e_1.\ e_1 \triangleright t_1 \text{ is } \mathbf{cf} \iff e_1 \in t_1 & \text{[IH1]} \\
\forall \mathbf{cl}\ e_2, \mathbf{cl\ tl}\ e'.\ e_2 \triangleright t_2[e'/x] \text{ is } \mathbf{cf} \iff e_2[e'/x] \in t_2[e'/x] & \text{[IH2]}
\end{array}
$$

We prove it as follows.

$$
\begin{array}{ll}
 & e \triangleright (x\colon t_1, t_2) \text{ is } \mathbf{cf} \\
\Longleftrightarrow & \text{(By defn of } \triangleright\text{)} \\
 & \texttt{match}\ e\ \texttt{with}\ \{(x_1, x_2) \to (x_1 \triangleright t_1, x_2 \triangleright t_2[x_1 \triangleleft t_1/x])\} \text{ is } \mathbf{cf} \\
\Longleftrightarrow & \text{(By [E-match] and defn of } \mathbf{cf}\text{)} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \text{ and } e_2 \text{ are } \mathbf{cf} \text{ and} \\
 & (e_1 \triangleright t_1) \text{ is } \mathbf{cf} \text{ and } (e_2 \triangleright t_2[e_1 \triangleleft t_1/x]) \text{ is } \mathbf{cf} \\
\Longleftrightarrow & \text{(By [IH1])} \\
(\dagger) & e \to^* (e_1, e_2) \text{ and } e_1 \text{ and } e_2 \text{ are } \mathbf{cf} \text{ and} \\
 & e_1 \in t_1 \text{ and } (e_2 \triangleright t_2[e_1 \triangleleft t_1/x]) \text{ is } \mathbf{cf}
\end{array}
$$

Now the proof splits into two. In the forward direction, we start with (†):

$$
\begin{array}{ll}
(\dagger) & e \to^* (e_1, e_2) \text{ and } e_1 \text{ and } e_2 \text{ are } \mathbf{cf} \text{ and} \\
 & e_1 \in t_1 \text{ and } e_2 \triangleright t_2[e_1 \triangleleft t_1/x] \text{ is } \mathbf{cf} \\
\Rightarrow & \text{(By Lemma } 16^{\text{p57}} \text{ (Conditional projection) (a) and} \\
 & \text{Theorem } 14^{\text{p55}} \text{ (congruence of } \preceq\text{) and} \\
 & \text{Lemma } 7^{\text{p15}} \text{ (c) (property of } \preceq\text{))} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } e_2 \triangleright t_2[e_1/x] \text{ is } \mathbf{cf} \\
\Longleftrightarrow & \text{(By [IH1] and [IH2])} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } e_2 \in t_2[e_1/x] \\
\Longleftrightarrow & \text{(By definition of } \in\text{)} \\
 & e \in (x\colon t_1, t_2)
\end{array}
$$

The key step is the use of Lemma 16^{p57} (a) (see Section A.6).

Now we prove the reverse direction. We use the fact that $(x\colon t_1, t_2)$ is total. By definition of total contract, t_1 is total and for all $e \in t_1$, $t_2[e/x]$ is total.

We have:

$$
\begin{array}{ll}
 & e \in (x\colon t_1, t_2) \\
\iff & \text{(By definition of } \in\text{)} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } e_2 \in t_2[e_1/x] \\
\iff & \text{(By Lemma } 14^{\text{p53}} \text{ (Key lemma), let } e_1 = e' \lhd t_1\text{)} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \in t_1 \text{ and } \exists \mathbf{cf}\ e', e_2 \in t_2[e' \lhd t_1/x] \\
\iff & \text{(By [IH1])} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \rhd t_1 \text{ is } \mathbf{cf} \text{ and } \exists \mathbf{cf}\ e', e_2 \in t_2[e' \lhd t_1/x] \\
\Rightarrow & (e_1 \rhd t_1 \text{ is } \mathbf{cf} \text{ and by [IH2]}) \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \rhd t_1 \text{ is } \mathbf{cf} \text{ and } e_2 \rhd t_2[e_1 \rhd t_1 \lhd t_1/x] \text{ is } \mathbf{cf} \\
\iff & \text{(By Lemma } 15^{\text{p56}} \text{ (Idempotency)} \\
 & \text{Theorem } 14^{\text{p55}} \text{ (congruence of } \preceq\text{) and} \\
 & \text{Lemma } 7^{\text{p15}} \text{ (c) (property of } \preceq\text{))} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \rhd t_1 \text{ is } \mathbf{cf} \text{ and } e_2 \rhd t_2[e_1 \rhd t_1 \lhd t_1 \lhd t_1/x] \text{ is } \mathbf{cf} \\
\iff & \text{(By Theorem } 15^{\text{p55}} \text{ (Projection pair), } e_1 \rhd t_1 \lhd t_1 \preceq e_1\text{,} \\
 & \text{Theorem } 14^{\text{p55}} \text{ (congruence of } \preceq\text{) and} \\
 & \text{Lemma } 7^{\text{p15}} \text{ (c) (property of } \preceq\text{))} \\
 & e \to^* (e_1, e_2) \text{ and } e_1 \rhd t_1 \text{ is } \mathbf{cf} \text{ and } e_2 \rhd t_2[e_1 \lhd t_1/x] \text{ is } \mathbf{cf}
\end{array}
$$

The key steps are using Lemma 14^{p53} (Key lemma), apply Lemma 15^{p56} (Idempotency) and use Theorem 15^{p55} (Projection pair).

- t is `Any`: We have:

$$
\begin{array}{ll}
 & e \rhd \texttt{Any} \text{ is } \mathbf{cf} \\
\iff & \text{(By definition of } \rhd\text{)} \\
 & \texttt{UNR} \text{ is } \mathbf{cf} \\
\iff & \text{(By definition of } \in\text{, and } \texttt{UNR} \in \texttt{Any}\text{)} \\
 & e \in \texttt{Any}
\end{array}
$$

□

A.1 Telescoping Property

The telescoping property is adapted from [7] and we found that this property makes the proofs of many lemmas shorter. However, it is not used in any proof in [7].

Lemma 13 (Telescoping Property)**.** *For all expression e, and total contract t,* $(e \underset{r_2}{\overset{r_1}{\bowtie}} t) \underset{r_4}{\overset{r_3}{\bowtie}} t = e \underset{r_4}{\overset{r_1}{\bowtie}} t$

Proof. Before we start the proof, by definition of `let`, [E-exn] and [E-match], we know two facts:

[Fact1] $\forall e'.\ (\texttt{let}\ x = \texttt{BAD}\ \texttt{in}\ e') \to \texttt{BAD}$

[Fact2] $\forall alts,\ (\texttt{match}\ \texttt{BAD}\ \texttt{with}\ alts) \to \texttt{BAD}$

The proof begins by dealing with two special cases.

- Case $e \to^* \texttt{BAD}$: Based on [Fact1] and [Fact2], for all $t \neq \texttt{Any}$, by inspecting the definition of $\bowtie$, we know $(e \overset{r_i}{\underset{r_j}{\bowtie}} t) \to^* \texttt{BAD}$ for all i, j. So LHS=RHS=BAD for $t \neq \texttt{Any}$. In the case $t = \texttt{Any}$, we have:

$$\begin{array}{ll} & (e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any}) \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\ = & r_2 \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\ = & r_4 \\ = & e \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \end{array}$$

- $e\uparrow$. Similar to the arguments in the case $e \to^* \texttt{BAD}$.

Hence for the rest of the proof we assume that $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$. The rest of the proof is by induction on the size of t.

- t is $\{x \mid p\}$:

$$\begin{array}{ll} & (e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\}) \overset{r_3}{\underset{r_4}{\bowtie}} \{x \mid p\} \\ = & \text{(By definition of } \bowtie) \\ & \texttt{let}\ x = (\ \texttt{let}\ x = e\ \texttt{in if}\ p\ \texttt{then}\ x\ \texttt{else}\ r_1\ ) \\ & \texttt{in if}\ p\ \texttt{then}\ x\ \texttt{else}\ r_3 \\ = & \text{(We float } \texttt{let}\ x = e \text{ out)} \\ & \texttt{let}\ x = e\ \texttt{in if}\ p\ \texttt{then}\ (\texttt{let}\ x = x\ \texttt{in if}\ p\ \texttt{then}\ x\ \texttt{else}\ r_3) \\ & \qquad\qquad\qquad\ \texttt{else}\ (\texttt{let}\ x = r_1\ \texttt{in if}\ p\ \texttt{then}\ x\ \texttt{else}\ r_3) \\ = & \text{(This is not } \texttt{let rec}\text{, so inline } x \text{ in the } \texttt{then} \text{ branch.} \\ & \text{By [E-beta] and [Fact1].)} \\ & \texttt{let}\ x = e\ \texttt{in if}\ p\ \texttt{then}\ (\texttt{if}\ p\ \texttt{then}\ x\ \texttt{else}\ r_3) \\ & \qquad\qquad\qquad\ \texttt{else}\ r_1 \\ = & \text{(propagating the } \texttt{true} \text{ value of } p \text{ to sub-branches)} \\ & \texttt{let}\ x = e\ \texttt{in if}\ p\ \texttt{then}\ x \\ & \qquad\qquad\qquad\ \texttt{else}\ r_1 \\ = & \text{(By defn of } \bowtie) \\ & e \overset{r_1}{\underset{r_4}{\bowtie}} t \end{array}$$

- t is $x\colon t_1 \to t_2$: We have the following induction hypotheses:

$$\forall e, \mathbf{tl}\ t_1,\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_1) \overset{r_3}{\underset{r_4}{\bowtie}} t_1 = e \overset{r_1}{\underset{r_4}{\bowtie}} t_1 \qquad \text{[IH1]}$$

$$\forall e, e' \in t_1, \mathbf{tl}\ t_2[e'/x],\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_2[e'/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[e'/x] = e \overset{r_1}{\underset{r_4}{\bowtie}} t_2[e'/x] \qquad \text{[IH2]}$$

We have the following proof:

$$
\begin{array}{ll}
 & (e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2) \overset{r_3}{\underset{r_4}{\bowtie}} x\colon t_1 \to t_2 \\
= & \text{(By defn of } \bowtie\text{)} \\
 & \texttt{let } y = e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 \texttt{ in } \lambda x_1.\ (y \quad (x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By defn of } \bowtie \text{ again)} \\
 & \texttt{let } y = e \texttt{ in let } y = \lambda x_2.\ ((y\ (x_2 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_2 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \texttt{ in} \\
 & \lambda x_1.\ ((y \quad (x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x]) \\
= & \text{(By } \beta\text{-reduction)} \\
 & \texttt{let } y = e \texttt{ in} \\
 & \lambda x_1.\ ((y\ ((x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1) \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By induction hypothesis with } t = t_1\text{)} \\
 & \texttt{let } y = e \texttt{ in } \lambda x_1.\ ((y\ (x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x] \\
= & \text{(By call-by-value, } r_i \text{ in } t_2 \text{ (for all } i\text{) are not reachable, replace } r_3 \text{ by } r_1\text{)} \\
 & \texttt{let } y = e \texttt{ in } \lambda x_1.\ ((y\ (x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
= & \text{(By induction hypothesis [IH2]: } t = t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
 & t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \text{ is } \mathbf{tl} \text{ because } r_i \text{ in } t_2 \text{ (for all } i\text{) are not reachable)} \\
 & \texttt{let } y = e \texttt{ in } \lambda x_1.\ (y\ (x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x] \\
= & \text{(By defn of } \bowtie\text{)} \\
 & e \overset{r_1}{\underset{r_4}{\bowtie}} x\colon t_1 \to t_2
\end{array}
$$

Although the β-reduction is done in the body of a `let`-expression, it is valid because we know $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$ and it does not violate call-by-value execution.

- t is $(x\colon t_1, t_2)$: We have the following induction hypotheses:

$$\forall e, \mathbf{tl}\ t_1,\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_1) \overset{r_3}{\underset{r_4}{\bowtie}} t_1 = e \overset{r_1}{\underset{r_4}{\bowtie}} t_1 \qquad \text{[IH1]}$$

$$\forall e, e' \in t_1, \mathbf{tl}\ t_2[e'/x],\ (e \overset{r_1}{\underset{r_2}{\bowtie}} t_2[e'/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[e'/x] = e \overset{r_1}{\underset{r_4}{\bowtie}} t_2[e'/x] \qquad \text{[IH2]}$$

We have the following proof:

$$
\begin{array}{ll}
 & (e \overset{r_1}{\underset{r_2}{\bowtie}} (x\colon t_1, t_2)) \overset{r_3}{\underset{r_4}{\bowtie}} (x\colon t_1, t_2) \\
= & \text{(By defn of } \bowtie\text{)} \\
 & (\texttt{match}\ e\ \texttt{with}\ (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{r_1}{\bowtie}} t_1)/x])) \overset{r_3}{\underset{r_4}{\bowtie}} (x\colon t_1, t_2) \\
= & \text{(By defn of } \bowtie \text{ again)} \\
 & \texttt{match}\ (\texttt{match}\ e\ \texttt{with}\ (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{r_1}{\bowtie}} t_1)/x]))\ \texttt{with} \\
 & (x_3, x_4) \to (x_3 \overset{r_3}{\underset{r_4}{\bowtie}} t_1, x_4 \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_3 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x]) \\
= & \text{(By simpl rule [match-match] and [E-match])} \\
 & \texttt{match}\ e\ \texttt{with} \\
 & (x_1, x_2) \to ((x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1) \overset{r_3}{\underset{r_4}{\bowtie}} t_1, (x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1 \overset{r_4}{\underset{r_3}{\bowtie}} t_1)/x]) \\
= & \text{(By induction hypothesis [IH1].} \\
 & \texttt{match}\ e\ \texttt{with} \\
 & (x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_4}{\bowtie}} t_1, (x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_1}{\underset{r_3}{\bowtie}} t_1)/x]) \\
= & \text{(Due to } x_1 \overset{r_1}{\underset{r_4}{\bowtie}} t_1\text{, for all } i, j\text{, the } r_i, r_j \text{ in } [x_1 \overset{r_i}{\underset{r_j}{\bowtie}} t_1/x] \text{ cannot be reached.)} \\
 & \texttt{match}\ e\ \texttt{with} \\
 & (x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_4}{\bowtie}} t_1, (x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \overset{r_3}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \\
= & \text{(By induction hypothesis [IH2]: } t = t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x].\text{)} \\
 & \texttt{match}\ e\ \texttt{with} \\
 & (x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_4}{\bowtie}} t_1, x_2 \overset{r_1}{\underset{r_4}{\bowtie}} t_2[(x_1 \overset{r_4}{\underset{r_1}{\bowtie}} t_1)/x]) \\
= & \text{(By defn of projection)} \\
 & e \overset{r_1}{\underset{r_4}{\bowtie}} (x\colon t_1, t_2)
\end{array}
$$

- t is `Any`:

  LHS

  $$
  \begin{array}{ll}
   & (e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any}) \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\
  = & r_2 \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\
  = & r_4
  \end{array}
  $$

  RHS

  $$
  \begin{array}{ll}
   & e \overset{r_3}{\underset{r_4}{\bowtie}} \texttt{Any} \\
  = & r_4
  \end{array}
  $$

  Since LHS $\equiv$ RHS, we are done.

□

A.2 Key Lemma

Lemma 14 (Key lemma)**.** *For all crash-free e and total contract t, such that $\vdash e :: \tau$ and $\vdash_c t :: \tau$,*

$$e \triangleleft t \ \in \ t$$

Proof. First, we have the following derivation (named D1).

$$
\begin{array}{ll}
 & (e \lhd t) \rhd t \\
= & \text{(By defn of } \lhd \text{ and } \rhd\text{)} \\
 & (e \underset{\text{BAD}}{\overset{\text{UNR}}{\bowtie}} t) \underset{\text{UNR}}{\overset{\text{BAD}}{\bowtie}} t \\
= & \text{(By Lemma 13}^{\text{p50}}\text{ (Telescoping Property))} \\
 & e \underset{\text{UNR}}{\overset{\text{UNR}}{\bowtie}} t
\end{array}
$$

Now, we have the following proof.

$$
\begin{array}{ll}
 & e \text{ is } \mathbf{cf} \\
\Rightarrow & \text{(Since } t \text{ is total, } t \equiv \lfloor t \rfloor\text{. By the defn of } \bowtie\text{, the context } (\bullet \underset{\text{UNR}}{\overset{\text{UNR}}{\bowtie}} \lfloor t \rfloor) \\
 & \text{ is syntactically safe. By defn of } \mathbf{cf}\text{, we have below)} \\
 & e \underset{\text{UNR}}{\overset{\text{UNR}}{\bowtie}} t \text{ is } \mathbf{cf} \\
\Longleftrightarrow & \text{(By derivation D1)} \\
 & (e \lhd t) \rhd t \text{ is } \mathbf{cf} \\
\Longleftrightarrow & \text{(By Theorem 2}^{\text{p18}}\text{ (grand theorem))} \\
 & (e \lhd t) \in t
\end{array}
$$

□

A.3 Examination of Cyclic Dependencies

Recall the dependency graph in Figure 18, there are two cycles:

(1) T2 → L15 → T2

(2) T2 → L17 → L19 → T2

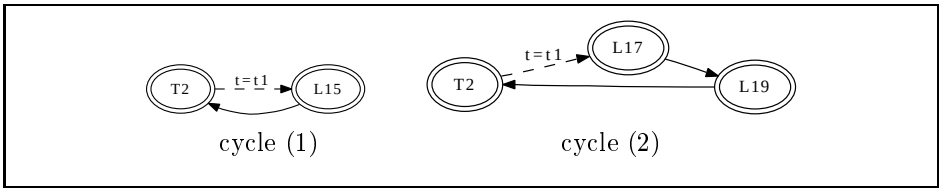

Figure 19: Cyclic Dependency of Three Lemmas

Each cycle is shown in Figure 19. The dashed directed edge indicates a decrease in size of t while the solid directed edge shows a preservation of the size of t. We can see that, in each cycle, there is an edge that decreases the size of t. Cycle (1) is well-founded because the size of t (where $t = x\colon t_1 \to t_2$) decreases (to t_1) when Theorem 2^{p18} calls Lemma 14^{p53}. Cycle (2) is well-founded because the size of t (where $t = x\colon t_1 \to t_2$) decreases (to t_1) when Theorem 2^{p18} calls Lemma 16^{p57}. Although there are cyclic dependencies among these theorems and lemmas, on each cyclic path, there is a decrease in the size of t. Thus, our proof on induction of the size of t is well-founded.

A.4 Congruence of Crashes-More-Often

Theorem 14 (Congruence of Crashes-More-Often)**.**

$$\forall e_1, e_2.\quad e_1 \preceq e_2 \quad\Longleftrightarrow\quad \forall \mathcal{C},\ \ \mathcal{C}[\![e_1]\!] \preceq \mathcal{C}[\![e_2]\!]$$

Proof. We prove two directions separately:

($\Rightarrow$) For an arbitrary $\mathcal{B}$, we prove $\mathcal{B}[\![e_1]\!] \preceq \mathcal{B}[\![e_2]\!]$. We have the following proof:

$$\begin{array}{ll} & e_1 \preceq e_2 \\ \Longleftrightarrow & \text{(By definition 7)} \\ & \forall \mathcal{C}.\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD} \\ \Rightarrow & \forall \mathcal{C}, \mathcal{D}.\ (\mathcal{C} = \mathcal{D}[\![\mathcal{B}[\![\bullet]\!]]\!]) \Rightarrow (\mathcal{C}[\![e_2]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* \texttt{BAD}) \\ \Rightarrow & \forall \mathcal{D}.\ \mathcal{D}[\![\mathcal{B}[\![e_2]\!]]\!] \rightarrow^* \texttt{BAD} \Rightarrow \mathcal{D}[\![\mathcal{B}[\![e_1]\!]]\!] \rightarrow^* \texttt{BAD} \\ \Rightarrow & \forall \mathcal{B}.\mathcal{B}[\![e_1]\!] \preceq \mathcal{B}[\![e_2]\!] \end{array}$$

Note that we assume for all $i = 1, 2$:

$$\vdash \mathcal{C}[\![e_i]\!] :: \texttt{bool},\ \vdash \mathcal{D}[\![e_i]\!] :: \texttt{bool} \text{ and } \vdash \mathcal{E}[\![e_i]\!] :: \texttt{bool}$$

($\Leftarrow$) It is trivially true, because we can choose an empty context (i.e. $\mathcal{C} = \bullet$). □

A.5 Projection Pair and Closure Pair

Recall the definition of projection pair. Let D and E be complete partial order's. If $f : D \to E$ and $g : E \to D$ are continuous functions such that $f \circ g \subseteq id$, then (f,g) is called a projection pair. If $id \subseteq f \circ g$, then (f,g) is called a closure pair. In this section, we are not going to explore the theory in depth. We only notice that in some way ($\bullet \triangleright t \triangleleft t \preceq id$) and ($id \preceq \bullet \triangleleft t \triangleright t$) match the definition of projection pair and closure pair respectively.

Theorem 15 (A projection pair)**.** *For all expression e and contract t, such that $\exists \Gamma.\ \Gamma \vdash e :: \tau$ and $\Gamma \vdash_c t :: \tau$,*

$$(e \triangleright t) \triangleleft t \preceq e$$

Proof. We have the following proof:

$$\begin{array}{ll} & (e \triangleright t) \triangleleft t \\ = & \text{(By defn of } \triangleright \text{ and } \triangleleft\text{)} \\ & (e \underset{\texttt{UNR}}{\overset{\texttt{BAD}}{\bowtie}} t) \underset{\texttt{BAD}}{\overset{\texttt{UNR}}{\bowtie}} t \\ = & \text{(By Lemma 13}^{\text{p50}}\text{)} \\ & e \underset{\texttt{BAD}}{\overset{\texttt{BAD}}{\bowtie}} t \\ \ll_{\{\texttt{BAD}\}} & \text{(By Lemma 19}^{\text{p58}}\text{)} \\ & e \end{array}$$

By definition of $\ll_{\{\texttt{BAD}\}}$, we get the desired result. □

Theorem 16 (A Closure Pair)**.** *For all expression e and contract t, such that $\exists\Gamma.\, \Gamma \vdash e :: \tau$ and $\Gamma \vdash_c t :: \tau$,*

$$e \preceq (e \lhd t) \rhd t$$

Proof. We have the following proof:

$$\begin{array}{rl} & (e \lhd t) \rhd t \\ = & \text{(By defn of } \lhd \text{ and } \rhd) \\ & (e \underset{\text{BAD}}{\overset{\text{UNR}}{\bowtie}} t) \underset{\text{UNR}}{\overset{\text{BAD}}{\bowtie}} t \\ = & \text{(By Lemma 13}^{\text{p50}}\text{)} \\ & e \underset{\text{UNR}}{\overset{\text{UNR}}{\bowtie}} t \\ \ll_{\{\text{UNR}\}} & \text{(By Lemma 19}^{\text{p58}}\text{)} \\ & e \end{array}$$

By definition of $\ll_{\{\text{UNR}\}}$, we get the desired result. □

A.6 Contracts are Projections

Recall the definition of projection, a projection p is a function that has two properties:

1. $p = p \circ p$
2. $p \subseteq 1$

The first one is called the retract property and says that projections are idempotent on their range. The second one says that the result of a projection contains no more information than its input.

We would like to show that if $e \in t$, then $(\bullet \lhd t)$ is an error projection while $(\bullet \rhd t)$ is a safe projection. By *error projection*, we mean $e \lhd t$ either behaves the same as e or returns `BAD`. Similarly, by *safe projection*, we mean $e \rhd t$ either behaves the same as e or returns `UNR`.

Findler and Blume [12] are the first to discover that contracts are pairs of projections. However, they assume that the e is a non-crashing term and the only error raised are contract violations. We assume that a program may contain errors and may crash. We give *error* a contract `Any`. Moreover, we prove different theorems from [12].

Theorem 17 (Error Projection)**.** *For all closed e and closed t, if $e \in t$, $(\bullet \lhd t)$ is a projection.*

Proof. By Lemma 15$^{\text{p56}}$ (a) (Idempotency) and Lemma 16$^{\text{p57}}$ (a). □

Theorem 18 (Safe Projection)**.** *For all closed e and closed t, if $e \in t$, $(\bullet \rhd t)$ is a projection.*

Proof. By Lemma 15$^{\text{p56}}$ (b) (Idempotency) and Lemma 16$^{\text{p57}}$ (b). □

Lemma 15 (Idempotence)**.** *For all closed e, t,*

$$e \underset{r_2}{\overset{r_1}{\bowtie}} t \underset{r_2}{\overset{r_1}{\bowtie}} t = e \underset{r_2}{\overset{r_1}{\bowtie}} t$$

Proof. It follows directly from Lemma 13[p50] (telescoping property). □

Lemma 16 (Conditional projection)**.** *For all closed e, closed and total t, if $e \in t$, then*

$$(a) \quad e \triangleleft t \preceq e \qquad\qquad (b) \quad e \preceq e \triangleright t$$

Proof. We prove each of them separately.

(a) Given $e \in t$, we have:

$$\begin{array}{ll} & e \triangleleft t \\ = & \text{(By defn of } \triangleright \text{ in Figure 6)} \\ & e \overset{\substack{\text{UNR}\\ \text{BAD}}}{\bowtie} t \\ \equiv_s & \text{(By Lemma 18[p57] (Exception III))} \\ & e \overset{\substack{\text{BAD}\\ \text{BAD}}}{\bowtie} t \\ \preceq & \text{(By Lemma 19[p58] (Behaviour of projection) and Definition 6[p14] } (\ll)) \\ & e \end{array}$$

(b) Given $e \in t$, we have:

$$\begin{array}{ll} & e \triangleright t \\ = & \text{(By defn of } \triangleright \text{ in Figure 6)} \\ & e \overset{\substack{\text{BAD}\\ \text{UNR}}}{\bowtie} t \\ \equiv_s & \text{(By Lemma 18[p57] (Exception III))} \\ & e \overset{\substack{\text{UNR}\\ \text{UNR}}}{\bowtie} t \\ \succeq & \text{(By Lemma 19[p58] (Behaviour of projection) and Definition 6[p14] } (\ll)) \\ & e \end{array}$$

□

Lemma 17 (Exception I)**.** $\forall \mathcal{C}.\ (\mathcal{C}[\![\texttt{UNR}, \texttt{BAD}]\!]$ *is* **cf** $\Rightarrow \forall r_1, r_2 \in \{\texttt{BAD}, \texttt{UNR}\}.\ \mathcal{C}[\![\texttt{UNR}, r_1]\!] \equiv_s \mathcal{C}[\![\texttt{UNR}, r_2]\!])$

Proof. The intuition is that the `BAD` in the hole cannot be reached, so we can replace it by any exceptional value. This reasoning in turn relies on the absence of a "catch" primitive that can transform `BAD` into something non-`BAD`.

Formally, we can prove the lemma by case splitting on whether $\mathcal{C}[\![\texttt{UNR}, \texttt{BAD}]\!]$ terminates, and if it does, by induction on the number of steps of reduction. □

Lemma 18 (Exception III)**.** $\forall e, t.\ e \in t \Rightarrow \forall r.\ e \overset{\substack{\text{BAD}\\ r}}{\bowtie} t \equiv_s e \overset{\substack{\text{UNR}\\ r}}{\bowtie} t$

Proof. For all expression e, contract t, we have:

$$\begin{array}{ll} & e \in t \\ \iff & \text{(By Theorem 2}^{\text{p18}}\text{ (Grand Theorem))} \\ & e \triangleright t \text{ is } \mathbf{cf} \\ \iff & \text{(By defn of } \triangleright \text{ and } \mathbf{cf}) \\ & \forall \mathcal{C},\ \mathtt{BAD} \notin \mathcal{C}.\ \mathcal{C}[\![e \overset{\text{BAD}}{\underset{r}{\bowtie}} t]\!] \not\to^* \mathtt{BAD} \\ \iff & \text{(By Lemma 17}^{\text{p57}}\text{ (Exception I))} \\ & \forall \mathcal{C},\ \mathtt{BAD} \notin \mathcal{C}.\ \mathcal{C}[\![e \overset{\text{BAD}}{\underset{r}{\bowtie}} t]\!] \equiv_s \mathcal{C}[\![e \overset{\text{UNR}}{\underset{r}{\bowtie}} t]\!] \\ \Rightarrow & \text{(Let } \mathcal{C} = \bullet) \\ & e \overset{\text{BAD}}{\underset{r}{\bowtie}} t \equiv_s e \overset{\text{UNR}}{\underset{r}{\bowtie}} t \end{array}$$

We are done. □

A.7 Behaviour of Projections

We have seen that in Section A.5, we make use of the property of behaves-the-same ($\ll$) (Lemma 19$^{\text{p58}}$). In this section, we give its detailed proof. Lemma 19$^{\text{p58}}$ says that an expression wrapped with a contract behaves either the same as the original expression or returns one of the exceptions which can be either `BAD` or `UNR`.

Lemma 19 (Behaviour of projection)**.** *For all r_1, r_2, e, total t, such that $\vdash e :: \tau$ and $\vdash_c t :: \tau$, and $r_1, r_2 \in \{\mathtt{BAD}, \mathtt{UNR}\}$,*

$$e \overset{r_1}{\underset{r_2}{\bowtie}} t \ll_{\{r_1, r_2\}} e$$

Proof. The proof begins by dealing with two special cases: $e\uparrow$, $e \to^* \mathtt{BAD}$. In both cases, by Definition of $\bowtie$, we know $e \overset{r_1}{\underset{r_2}{\bowtie}} t \equiv_s e$ and we are done.

Hence, for the rest of the proof we assume that $e \to^* val \notin \{\mathtt{BAD}, \mathtt{UNR}\}$. We prove it by induction on the size of t. Let R be $\{r_1, r_2\}$.

- t is $\{x \mid p\}$: we have

$$\begin{array}{rcl} e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} & = & \mathtt{let}\ x = e\ \mathtt{in}\ \ \mathtt{match}\ p[e/x]\ \mathtt{with} \\ & & \quad \mid \mathtt{true} \to e \\ & & \quad \mid \mathtt{false} \to r_1 \end{array}$$

  Since t is total, $p[e/x] \not\to^* \mathtt{BAD}$. So there are two cases to consider:

  - If $p[e/x] \to^* \mathtt{false}$, then $e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \to^* r_1$ and we are done.
  - If $p[e/x] \to^* \mathtt{true}$, $e \overset{r_1}{\underset{r_2}{\bowtie}} \{x \mid p\} \to^* e$ and we are done.

- t is $x\colon t_1 \to t_2$: We have

$$e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2 \quad = \quad \mathtt{let}\ y = e\ \mathtt{in}\ \lambda v.\ ((y\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x])$$

Since $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$, $e \to^* \lambda x.e'$ and $(e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2) \to^* \lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x])$.

We want to show that $\forall \mathcal{C}.\ \mathcal{C}[\![e]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![\lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]]\!]) \to^* r$. We prove it by induction on contexts. There are 3 cases to consider:

1. $\mathcal{C} = [\![\bullet]\!]$;
2. $\mathcal{C} = \mathcal{D}[\![\texttt{match}\ \bullet\ \texttt{with}\ alts]\!]$;
3. $\mathcal{C} = \mathcal{D}[\![\bullet\ e_3]\!]$.

Case 1 and 2 are trivially true by inspecting the operational semantics of `match`. For Case 3, since we prove it by induction on the size of context, we have the following induction hypothesis:

$$\forall \mathcal{D}[\![e]\!] \to^* r \Rightarrow \mathcal{D}[\![\bullet\ e_3]\!] \to^* r \qquad \text{[IH]}$$

So all we need to prove is that for all e_3,

$$(\lambda v.\ ((e\ (v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(v \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]))\ e_3 \ll_R e\ e_3$$

By β-reduction, it means we want to show

$$(e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x] \ll_R (e\ e_3) \qquad (*)$$

By induction hypotheis where $t = t_2[(e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x]$, we have

$$(e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x] \ll_R (e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)) \quad (1)$$

By induction hypothesis where $t = t_1$, we have

$$e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1 \ll_R e_3$$

By Lemma 20^{p60} (Congruence of $\ll_R$), we have

$$e\ (e_3 \overset{r_2}{\underset{r_1}{\bowtie}} t_1) \ll_R e\ e_3 \qquad (2)$$

By (1) and (2) and Lemma 21^{p60} (Transitivity of $\ll_R$), we get (*). By [IH], we have the desired result $\forall \mathcal{C}.\ \mathcal{C}[\![e]\!] \to^* r \in R \Rightarrow \mathcal{C}[\![e \overset{r_1}{\underset{r_2}{\bowtie}} x\colon t_1 \to t_2]\!] \to^* r$.

- t is $(x\colon t_1, t_2)$: We have

$$e \overset{r_1}{\underset{r_2}{\bowtie}} (t_1, t_2) \quad = \quad \begin{array}{l}\texttt{match}\ e\ \texttt{with} \\ \quad (x_1, x_2) \to (x_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1,\ x_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2[(x_1 \overset{r_2}{\underset{r_1}{\bowtie}} t_1)/x])\end{array}$$

If $e \to^* val \notin \{\texttt{BAD}, \texttt{UNR}\}$, then $e \to^* \{e_1, e_2\}$. By the induction hypotheses where $t = t_1$ and $t = t_2$ respectively, we know $e_1 \overset{r_1}{\underset{r_2}{\bowtie}} t_1 \ll_R e_1$ and $e_2 \overset{r_1}{\underset{r_2}{\bowtie}} t_2 \ll_R e_2$. Therefore, by Definition 6^{p14}, we have $e \overset{r_1}{\underset{r_2}{\bowtie}} (t_1, t_2) \ll_R e$.

- t is `Any`: Since we have $e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any} = r_2$, we know $e \overset{r_1}{\underset{r_2}{\bowtie}} \texttt{Any} \rightarrow^* r_2$. By Definition 6^{p14}, we are done.

□

Lemma 20 (Congruence of Behaves-the-same)**.** *If* $e_1 \ll_R e_2$*, then* $\forall \mathcal{C}$, $\mathcal{C}[\![e_1]\!] \ll_R \mathcal{C}[\![e_2]\!]$.

Proof. we have the following proof:

$$\begin{array}{ll} & e_1 \ll_R e_2 \\ \iff & \text{(By definition 6)} \\ & \forall \mathcal{C},\ \mathcal{C}[\![e_2]\!] \rightarrow^* r \in R \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* r \\ \Rightarrow & \text{(Choose } \mathcal{C} \text{ be } \mathcal{D}[\![\mathcal{C}[\![E]\!]\bullet]\!]) \\ & \forall \mathcal{D}, \forall \mathcal{E},\ \mathcal{D}[\![\mathcal{E}[\![e_2]\!]]\!] \rightarrow^* r \in R \Rightarrow \mathcal{D}[\![\mathcal{E}[\![e_1]\!]]\!] \rightarrow^* r \\ \iff & \text{(By definition 6)} \\ & \forall \mathcal{C},\ \mathcal{C}[\![e_1]\!] \ll_R \mathcal{C}[\![e_2]\!] \end{array}$$

Note that we assume for all $i = 1, 2$:

$$\vdash \mathcal{C}[\![e_i]\!] :: (), \vdash \mathcal{D}[\![e_i]\!] :: () \text{ and } \vdash \mathcal{E}[\![e_i]\!] :: ()$$

□

Lemma 21 (Transitivity of $\ll_R$)**.** *If* $e_1 \ll_R e_2$ *and* $e_2 \ll_R e_3$*, then* $e_1 \ll_R e_3$.

Proof. By Definition 6^{p14}, we have

$$\begin{array}{ll} (1) & \forall \mathcal{C}.\ \mathcal{C}[\![e_2]\!] \rightarrow^* r \in R \Rightarrow \mathcal{C}[\![e_1]\!] \rightarrow^* r \\ (2) & \forall \mathcal{C}.\ \mathcal{C}[\![e_3]\!] \rightarrow^* r \in R \Rightarrow \mathcal{C}[\![e_2]\!] \rightarrow^* r \end{array}$$

For all $\mathcal{C}$, assuming $\mathcal{C}[\![e_3]\!] \rightarrow^* r \in R$, we want to show $\mathcal{C}[\![e_1]\!] \rightarrow^* r$. We have the following proof:

$$\begin{array}{ll} \forall \mathcal{C}. & \mathcal{C}[\![e_3]\!] \rightarrow^* r \in R \\ \Rightarrow & \text{(By (2))} \\ & \mathcal{C}[\![e_2]\!] \rightarrow^* r \in R \\ \Rightarrow & \text{(By (1))} \\ & \mathcal{C}[\![e_1]\!] \rightarrow^* r \end{array}$$

□

B Correctness of SL machine

B.1 Correctness of Logicization

Theorem 11 (Logicization for axioms) Given a definition $f = e^\tau$, the logical formula $\forall fv(e), \exists f : \tau.[\![e]\!]_f$ is valid.

Proof. We prove it by structural induction on the size of the (possiblly open) expression e. As `UNR` is for internal usage, we do not have `UNR` in e.

- Case e is $\texttt{BAD}^l$. We have $[\![\texttt{BAD}^l]\!]_f = \mathit{true}$, which is valid.

- Case e is x. We have $\exists f.f = x$. Let f be x, we have $x = x$, which is valid.
- Case e is n. We have $\exists f.f = n$. Let f be n, we have $n = n$, which is valid.
- Case e is $e_1^\tau \oplus e_2^\tau$. It is semantically equivalent to $\texttt{let}\ x_1 = e_1\ \texttt{in}\ \texttt{let}\ x_2 = e_2\ \texttt{in}\ x_1 \oplus x_2$. From $x_1 = e_1$, by induction hypothesis, (1) $[\![e_1]\!]_{x_1}$ is valid. From $x_2 = e_1$, by induction hypothesis, (2) $[\![e_2]\!]_{x_2}$ is valid. Let the existentially quantified f be $x_1 \oplus x_2$, we have (3) $x_1 \oplus x_2 = x_1 \oplus x_2$. From (1), (2), (3), we know $\exists f : \tau.\exists x_1 : [\![\tau]\!],\ \exists x_2 : [\![\tau]\!],\ ([\![e_1]\!]_{x_1} \wedge [\![e_2]\!]_{x_2} \wedge f = x_1 \oplus x_2)$ is valid.
- Case e is $e_1^\tau \odot e_2^\tau$. It is semantically equivalent to $\texttt{let}\ x_1 = e_1\ \texttt{in}\ \texttt{let}\ x_2 = e_2\ \texttt{in}\ x_1 \odot x_2$. From $x_1 = e_1$, by induction hypothesis, (1) $[\![e_1]\!]_{x_1}$ is valid. From $x_2 = e_1$, by induction hypothesis, (2) $[\![e_2]\!]_{x_2}$ is valid. If $e_1^{\tau_1} \odot e_2^{\tau_2}$ evaluates to $\texttt{true}$, $x_1 \odot x_2$ is valid and $not(x_1 \odot x_2)$ is invalid. So $\exists f : \tau, \exists x_1 : [\![\tau]\!],\ [\![e_1]\!]_{x_1} \wedge \exists x_2 : [\![\tau]\!],\ [\![e_2]\!]_{x_2} \wedge ((x_1 \odot x_2 \wedge f = \texttt{true}) \vee (not(x_1 \odot x2) \wedge f = \texttt{false}))$ deduces to $\exists f : \tau, \exists x_1 : [\![\tau_1]\!],\ [\![e_1]\!]_{x_1} \wedge \exists x_2 : [\![\tau_2]\!],\ [\![e_2]\!]_{x_2} \wedge (x_1 \odot x_2 \wedge f = \texttt{true})$. Let the existentially quantified f be $\texttt{true}$. From (1), (2) and $\texttt{true} = \texttt{true}$, we know $[\![e_1^\tau \odot e_2^\tau]\!]_f$ is valid. If $e_1^{\tau_1} \odot e_2^{\tau_2}$ evaluates to $\texttt{false}$, we apply the similar reasoning as above with the exitentially quantified f being $\texttt{false}$.
- Case e is $\lambda x^{\tau_1}.e_2^{\tau_2}$. We have $\exists f : \tau_1 \to \tau_2, \forall x : [\![\tau]\!],\ [\![e]\!]_{(\texttt{apply}(f,x))}$. Let the existentially quantified f be $\lambda x.e_2$.
- Case e is $\texttt{let}\ x^{\tau_1} = e_1\ \texttt{in}\ e_2^{\tau_2}$. It is semantically equivalent to $\texttt{let}\ x^{\tau_1} = e_1\ \texttt{in}\ \texttt{let}\ x_2^{\tau_2} = e_2\ \texttt{in}\ x_2$. We have $[\![\texttt{let}\ x^{\tau_1} = e_1\ \texttt{in}\ \texttt{let}\ x_2^{\tau_2} = e_2\ \texttt{in}\ x_2]\!]_f = \exists x : [\![\tau]\!],\ [\![e_1]\!]_x\ \wedge \exists x_2 : [\![\tau]\!], [\![e_2]\!]_{x_2} \wedge f = x_2$. From definitions $x^\tau = e_1$ and $x_2^{\tau_2} = e_2$, by induction hypothesis, (1) $\exists x : \tau_1, [\![e_1]\!]_x$ is valid and (2) $\exists x_2 : \tau_2, [\![e_2]\!]_{x_2}$ is valid. Let x_2 be f. From (1), (2) and $f = f$, we know $\exists f : \tau_2, \exists x : [\![\tau]\!],\ [\![e_1]\!]_x\ \wedge \exists x_2 : [\![\tau]\!], [\![e_2]\!]_{x_2} \wedge f = x_2$ is valid.
- Case e is $(e_1^{\tau_1}\ e_2^{\tau_2})$. It is semantically equivalent to $\texttt{let}\ x_1 = e_1\ \texttt{in}\ \texttt{let}\ x_2 = e_2\ \texttt{in}\ x_1\ x_2$. We have $[\![\texttt{let}\ x_1 = e_1\ \texttt{in}\ \texttt{let}\ x_2 = e_2\ \texttt{in}\ x_1\ x_2]\!]_f = \exists x_1 : \tau_1, [\![e_1]\!]_{x_1} \wedge \exists x_2 : \tau_2, [\![e_2]\!]_{x_2} \wedge f = apply(x_1, x_2)$. From definitions $x_1 = e_1$ and $x_2 = e_2$, by induction hypothesis, (1) $\exists x_1 : \tau_1, [\![e_1]\!]_{x_1}$ is valid and (2) $\exists x_2 : \tau_2, [\![e_2]\!]_{x_2}$ is valid. Let the existentially quantified f be $apply(x_1, x_2)$. From (1), (2) and $apply(x_1, x_2) = apply(x_1, x_2)$, we know $\exists x_1 : \tau_1, [\![e_1]\!]_{x_1} \wedge \exists x_2 : \tau_2, [\![e_2]\!]_{x_2} \wedge f = apply(x_1, x_2)$ is valid.
- Case e is $K^\tau\ e_1^{\tau_1} \ldots e_n^{\tau_n}$. It is semantically equivalent to $\texttt{let}\ x_1 = e_1\ \texttt{in} \ldots \texttt{let}\ x_n = e_n\ \texttt{in}\ K\ x_1 \ldots x_n$. We have $[\![\texttt{let}\ x_1 = e_1\ \texttt{in} \ldots \texttt{let}\ x_n = e_n\ \texttt{in}\ K\ x_1 \ldots x_n]\!]_f = \exists x_1 : \tau_1, [\![e_1]\!]_{x_1} \wedge \cdots \wedge \exists x_1 : \tau_1, [\![e_1]\!]_{x_1} \wedge f = K(x_1, \ldots, x_n)$. From definitions $x_i = e_i$ for $1 \leq i \leq n$, by induction hypothesis, we know (i) $\exists x_i.[\![e_i]\!]_{x_i}$ is valid. Let f be $K(x_1, \ldots, x_n)$. From (i) and $K(x_1, \ldots, x_n) = K(x_1, \ldots, x_n)$, we know $\exists f : \tau, \exists x_1 : \tau_1, [\![e_1]\!]_{x_1} \wedge \cdots \wedge \exists x_1 : \tau_1, [\![e_1]\!]_{x_1} \wedge f = K(x_1, \ldots, x_n)$ is valid.
- Case e is $\texttt{match}\ e_0^{\tau_0}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^{\tau_x}} \to e^\tau}$. It is semantically equivalent to $\texttt{let}\ x_0^{\tau_0} = e_0\ \texttt{in}\ \texttt{match}\ x_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^{\tau_x}} \to \texttt{let}\ y = e\ \texttt{in}\ y}$. We have $[\![\texttt{let}\ x_0^{\tau_0} = e_0\ \texttt{in}\ \texttt{match}\ x_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^{\tau_x}} \to \texttt{let}\ y = e\ \texttt{in}\ y}]\!]_f = \exists x_0 : \tau_0, [\![e_0]\!]_{x_0} \wedge$

$(\bigwedge \overrightarrow{\forall x : [\![\tau]\!]}, (x_0 = K\ \overrightarrow{x}) \Rightarrow \exists y : \tau, [\![e]\!]_y \wedge f = y)$. From definitions $x_0 = e_0$ and $y = e$, by induction hypothesis, (1) $\exists x_0 : \tau_0, [\![e_0]\!]_{x_0}$ is valid and (2) $\exists y : \tau, [\![e]\!]_y$ is valid. Let y be f. From (2) and $f = f$, the RHS of $\Rightarrow$ in the logical formula is valid. Together with (1), we know $\exists x_0 : \tau_0, [\![e_0]\!]_{x_0} \wedge (\bigwedge \overrightarrow{\forall x : [\![\tau]\!]}, (x_0 = K\ \overrightarrow{x}) \Rightarrow \exists y : \tau, [\![e]\!]_y \wedge f = y)$ is valid.

□

Theorem 12 (Logicization for goals: validity preservation) For all (possibly open) expression e^τ, if $\exists f : \tau, [\![e]\!]_f$ is valid and $e \to e'$ for some e', then $[\![e']\!]_f$ is valid.

Proof. We prove it by structural induction on the size of e. The lemma holds vacuriously for expressions `BAD`, `UNR`, x, n, $e_1 \oplus e_2$. We focus on two cases where a redex occurs. The rest of the cases can be proved easily by applying induction hypotheses.

- Case e is $(\lambda x^\tau.e_1)\ e_2$. We have

  $[\![(\lambda x^\tau.e_1)^{\tau_1}\ e_2^{\tau_2}]\!]_f$ is valid
  $\iff$ (By definition of $[\![.]\!]_f$)
  $\exists x_1 : [\![\tau_1]\!], [\![(\lambda x^\tau.e_1)]\!]_{x_1} \wedge \exists x_2 : [\![\tau_2]\!], [\![e_2]\!]_{x_2} \wedge$
  $f = apply(x_1, x_2)$ is valid
  $\iff$ (By definition of $[\![.]\!]_{x_1}$)
  $\exists x_1 : [\![\tau_1]\!], \forall x^\tau, [\![e_1]\!]_{(\texttt{apply}(x_1,x))} \wedge \exists x_2 : [\![\tau_2]\!], [\![e_2]\!]_{x_2} \wedge$
  $f = \texttt{apply}(x_1, x_2)$ is valid
  $\iff$ (By Logic: $P \wedge \exists x, Q(x) \iff \exists x, P \wedge Q(x)$ where x is not in P)
  $\exists x_1 : [\![\tau_1]\!], \exists x_2 : [\![\tau_2]\!], \forall x^\tau, [\![e_1]\!]_{(\texttt{apply}(x_1,x))} \wedge [\![e_2]\!]_{x_2} \wedge$
  $f = \texttt{apply}(x_1, x_2)$ is valid
  $\Rightarrow$ (Let x be x_2)
  $\exists x_1 : [\![\tau_1]\!], \exists x_2 : [\![\tau_2]\!], [\![e_1]\!]_{(\texttt{apply}(x_1,x_2))}[x_2/x] \wedge [\![e_2]\!]_{x_2} \wedge$
  $f = \texttt{apply}(x_1, x_2)$ is valid
  $\iff$ (Since $f = \texttt{apply}(x_1, x_2)$, replace $\texttt{apply}(x_1, x_2)$ by f)
  $\exists x_1 : [\![\tau_1]\!], \exists x_2 : [\![\tau_2]\!], [\![e_1]\!]_f[x_2/x] \wedge [\![e_2]\!]_{x_2}$ is valid
  $\iff$ (Rename x_2 to x)
  $\exists x_1 : [\![\tau_1]\!], \exists x : [\![\tau_2]\!], [\![e_1]\!]_f \wedge [\![e_2]\!]_x$ is valid
  $\iff$ (By Logic: $\exists x, P \iff P$ where x is not in P)
  $\exists x : [\![\tau_2]\!], [\![e_1]\!]_f \wedge [\![e_2]\!]_x$ is valid
  $\iff$ (By definition of $[\![.]\!]_f$)
  $[\![\texttt{let}\ x = e_2\ \texttt{in}\ e_1]\!]_f$ is valid
  $\iff$ ($\texttt{let}\ x = e_2\ \texttt{in}\ e_1$ is semantically equivalent to $e_1[e_2/x]$)
  $[\![e_1[e_2/x]]\!]_f$ is valid

- Case e is $\texttt{match}\ K\ \overrightarrow{a_i}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x}} \to e_i$. We have

$$
\begin{array}{ll}
 & [\![\texttt{match}\ (K\ \overrightarrow{val})^{\tau_0}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i}]\!]_f \text{ is valid} \\
\iff & \text{(By definition of } [\![.]\!]_f) \\
 & \exists x_0 : [\![\tau_0]\!], [\![K\ \overrightarrow{val}]\!]_{x_0} \wedge (\bigwedge \overrightarrow{\forall x : [\![\tau]\!], (x_0 = K\ \overrightarrow{x}) \Rightarrow [\![e_i]\!]_f}) \text{ is valid} \\
\iff & \text{(By definition of } [\![.]\!]_{x_0}) \\
 & \exists x_0 : [\![\tau_0]\!], \overrightarrow{\exists y : [\![\tau]\!], [\![val]\!]_y \wedge x_0 = K\ \overrightarrow{y}} \wedge \\
 & \overrightarrow{(\bigwedge \forall x : [\![\tau]\!], (x_0 = K\ \overrightarrow{x}) \Rightarrow [\![e_i]\!]_f)} \text{ is valid} \\
\Rightarrow & \text{(Let } \overrightarrow{x} \text{ be } \overrightarrow{y}) \\
 & \exists x_0 : [\![\tau_0]\!], \overrightarrow{\exists y : [\![\tau]\!], [\![val]\!]_y \wedge x_0 = K\ \overrightarrow{y}} \wedge \\
 & \overrightarrow{(\bigwedge (x_0 = K\ \overrightarrow{y}) \Rightarrow [\![e_i]\!]_f[\overrightarrow{y/x}])} \text{ is valid} \\
\iff & \text{(By Logic: } P \wedge (P \Rightarrow Q) \wedge (\neg P \Rightarrow R) \iff P \wedge Q) \\
 & \exists x_0 : [\![\tau_0]\!], \overrightarrow{\exists y : [\![\tau]\!], [\![val]\!]_y \wedge x_0 = K\ \overrightarrow{y} \wedge [\![e_i]\!]_f[\overrightarrow{y/x}]} \text{ is valid} \\
\Rightarrow & \text{(By Logic: } \exists x, \exists y, P(y) \wedge P(x,y) \iff \exists y, P(y) \wedge \exists x, P(x,y)) \\
 & \overrightarrow{\exists y : [\![\tau]\!], [\![val]\!]_y} \wedge \exists x_0 : [\![\tau_0]\!], x_0 = K\ \overrightarrow{y} \wedge [\![e_i]\!]_f[\overrightarrow{y/x}] \text{ is valid} \\
\iff & \text{(Let } x_0 \text{ be } K\ \overrightarrow{y}\text{. By Logic: } \texttt{true} \wedge P \iff P) \\
 & \overrightarrow{\exists y : [\![\tau]\!], [\![val]\!]_y} \wedge [\![e_i]\!]_f[\overrightarrow{y/x}] \text{ is valid} \\
\iff & \text{(Rename } \overrightarrow{y} \text{ to } \overrightarrow{x}) \\
 & \overrightarrow{\exists x : [\![\tau]\!], [\![val]\!]_x} \wedge [\![e_i]\!]_f \text{ is valid} \\
\iff & \text{(By definition of } [\![.]\!]_f) \\
 & [\![\texttt{let}\ \overrightarrow{x = val}\ \texttt{in}\ e_i]\!]_f \\
\iff & (\texttt{let}\ \overrightarrow{x = val}\ \texttt{in}\ e_i \text{ is semantically equivalent to } e_i[\overrightarrow{val/x}]) \\
 & [\![e_i[\overrightarrow{val/x}]]\!]_f \text{ is valid}
\end{array}
$$

□

B.2 Transition rules

The SL machine does not inline top-level functions. We do not have local `let rec` in our language and we only inline trivial values. Moreover, we set a stop-bound for the SMT solver Alt-ergo with an option "-stop <n>" (which restrict the total amount of time) or "-steps <n>" (which restrict the total number of steps) so that the SMT solver always terminates. Thus, there is no element in the SL machine causing non-termination.

Theorem 8 (SL machine terminates) For all $\mathcal{H}, e, \mathcal{S}, \mathcal{L}$, there exists an expression a such that $\langle \mathcal{H} \mid e \mid S \mid lgc \rangle \rightsquigarrow^* a$.

Proof. The rebuilding rules either lead to the end state ([R-done]) or reduce the number of stack frames ([R-r], [R-lam], [R-beta], [R-app], [R-K], [R-K-match], [R-s-match], [R-s-save]) or reduce the size of the stack frame on top of the stack ([R-fun]. [R-match], [R-let-save]).

The simplification rules either lead directly to a rebuild rule ([R-const], [R-exn], [R-var1], [R-var2]) or lead to a simplification rule that reduces the size of the expression under simplification ([S-lam], [S-app], [S-match], [S-K]) or lead to a simplification rule that reduces the size of the stack ([S-letL], [S-matchL], [S-letR], [S-matchR], [S-match-match], [S-match-let]). □

For the cases that corresponding to simplification rules in Figure 11, we use the fact: [EqFact] $e_1 \equiv_s e_2$ if $\exists e_3, e_1 \to^* e_3$ and $e_2 \to^* e_3$. Moreover, if any of the subexpression is an exception r, it is easy to show that both sides evaluate to the same r. So we only consider the case that none of the subexpression is an exception r.

Theorem 9 (Correctness of SL machine) For all closed expression e, if $\langle \emptyset \mid e \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* a$, then $e \equiv_s a$.

Proof. We prove it by induction on the number of transition steps. We have the following induction hypothesis: for all $\mathcal{H}, e, \mathcal{S}, \mathcal{L}$, there exists $\mathcal{H}_2, e_2, \mathcal{S}_2, \mathcal{L}_2$, such that $\langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle \mathcal{H}_2 \mid e_2 \mid \mathcal{S}_2 \mid \mathcal{L}_2 \rangle$ or $\langle \mathcal{H} \mid e \mid \mathcal{S} \mid \mathcal{L} \rangle \rightsquigarrow \langle\langle \mathcal{H}_2 \mid e_2 \mid \mathcal{S}_2 \mid \mathcal{L}_2 \rangle\rangle$,

$$\langle \mathcal{H}_2 \mid e_2 \mid \mathcal{S} \mid \mathcal{L}_2 \rangle \rightsquigarrow^* a \wedge e_2 \equiv_s a \quad \text{[IH]}$$

By Lemma 22[p68] (Correctness of rebuilding), we know

$$\langle\langle \mathcal{H}_2 \mid e_2 \mid \mathcal{S}_2 \mid \mathcal{L}_2 \rangle\rangle \rightsquigarrow^* a \wedge e_2 \equiv_s a \quad \text{[RB]}$$

For cases [S-const], [S-exn], [S-var1] [S-var2], by induction hybothesis, we get the desired result. We now focus on slightly non-obvious transitions.

- Case [S-lam]. We first have:

$$\begin{array}{ll} & \langle \mathcal{H} \mid \lambda x^\tau.e \mid [\,] \mid \emptyset \rangle \\ \rightsquigarrow & \text{(By [S-lam])} \\ & \langle \mathcal{H} \mid e \mid (\lambda x.\bullet) :: [\,] \mid \forall x : \tau \rangle \\ \rightsquigarrow^* & \text{(By [IH], } \langle \mathcal{H} \mid e \mid (\lambda x.\bullet) :: S \mid \mathcal{L}, \forall x : \tau \rangle \rightsquigarrow^* a \wedge e \equiv_s a\text{)} \\ & \langle\langle \mathcal{H} \mid a \mid (\lambda x.\bullet) :: [\,] \mid \forall x : \tau \rangle\rangle \\ \rightsquigarrow & \text{(By [R-lam])} \\ & \langle\langle \mathcal{H} \mid \lambda x.a \mid [\,] \mid \forall x : \tau \rangle\rangle \\ \rightsquigarrow & \text{(By [R-done])} \\ & \lambda x.a \end{array}$$

  We now have:

$$\begin{array}{ll} & e \equiv_s a \\ \iff & \text{(By Definition 1}^{\text{p11}}\text{ } \equiv_s\text{)} \\ & \forall \mathcal{C}, r, \mathcal{C}[\![e]\!] \to^* r \iff \mathcal{C}[\![a]\!] \to^* r \\ \iff & (\mathcal{C} = \mathcal{D}[\![\lambda x.\bullet]\!]) \\ & \forall \mathcal{D}, r, \mathcal{D}[\![\lambda x.e]\!] \to^* r \iff \mathcal{D}[\![\lambda x.a]\!] \to^* r \\ \iff & \text{(By Definition 1}^{\text{p11}}\text{ } \equiv_s\text{)} \\ & \lambda x.e \equiv_s \lambda x.a \end{array}$$

- Case [S-app]. If e_1 is r, it is easy. By [S-app] and [R-r-fun], we get $\langle \mathcal{H} \mid r\ e_2 \mid [\,] \mid \emptyset \rangle \rightsquigarrow^* r$, which is semantically equivalent to $r\ e_2$. We now consider the case where e_1 is not r. We have:

$$
\begin{array}{ll}
 & \langle \mathcal{H} \mid e_1\ e_2 \mid [\,] \mid \emptyset \rangle \\
\rightsquigarrow & \text{(By [S-app])} \\
 & \langle \mathcal{H} \mid e_1 \mid (\bullet\ e_2) :: [\,] \mid \emptyset \rangle \\
\rightsquigarrow^* & \text{(By [IH], } \langle \mathcal{H} \mid e_1 \mid (\bullet\ e_2) :: [\,] \mid \emptyset \rangle \rightsquigarrow^* a_1 \wedge e_1 \equiv_s a_1\text{)} \\
 & \langle\!\langle \mathcal{H} \mid a_1 \mid (\bullet\ e_2) :: [\,] \mid \emptyset \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-fun])} \\
 & \langle \mathcal{H} \mid e_2 \mid (a_1\ \bullet) :: [\,] \mid \emptyset \rangle \\
\rightsquigarrow^* & \text{(By [IH], } \langle \mathcal{H} \mid e_2 \mid (a_1\ \bullet) :: [\,] \mid \emptyset \rangle \rightsquigarrow^* a_2 \wedge e_2 \equiv_s a_2\text{)} \\
 & \langle\!\langle \mathcal{H} \mid a_2 \mid (a_1\ \bullet) :: [\,] \mid \emptyset \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-app])} \\
 & \langle\!\langle \mathcal{H} \mid a_1\ a_2 \mid [\,] \mid \emptyset \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-done])} \\
 & a_1\ a_2
\end{array}
$$

  Given $e_1 \equiv_s a_1$ and $e_2 \equiv_s a_2$, by congruence of $\equiv_s$, we know $e_1\ e_2 \equiv_s a_1\ a_2$.

- Case [S-match].

$$
\begin{array}{ll}
 & \langle \mathcal{H} \mid \texttt{match}\ e_0\ \texttt{with}\ alts \mid [\,] \mid \emptyset \rangle \\
\rightsquigarrow & \text{(By [S-match])} \\
 & \langle \mathcal{H} \mid e_0 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: [\,] \mid \emptyset \rangle \\
\rightsquigarrow^* & \text{(By [IH], } \langle \mathcal{H} \mid e_0 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: [\,] \mid \emptyset \rangle \rightsquigarrow^* a_0 \wedge e_0 \equiv_s a_0\text{)} \\
(\dagger) & \langle\!\langle \mathcal{H} \mid a_0 \mid (\texttt{match}\ \bullet\ \texttt{with}\ alts) :: [\,] \mid \emptyset \rangle\!\rangle
\end{array}
$$

  There are two subcases: either [R-s-match] or [R-s-save] is applied. Let $alts$ be $\overrightarrow{K\ \overrightarrow{x^?} \to e_i}$.

  - there exists a branch $(K\ \overrightarrow{x^?})$ such that $\mathcal{L} \Rightarrow (\overrightarrow{\exists x : [\![\tau]\!]}, [\![a_0]\!]_{(K\ \vec{x})})$. We continue from (†):

$$
\begin{array}{ll}
 & \langle\!\langle \mathcal{H} \mid a_0 \mid (\texttt{match}\ \bullet\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^?} \to e_i}) :: [\,] \mid \emptyset \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-s-match])} \\
 & \langle \mathcal{H} \mid e_i \mid [\,] \mid \exists \overrightarrow{x^?}, [\![a_0]\!]_{K\ \vec{x}} \rangle \\
\rightsquigarrow^* & \text{(By [IH], } \langle \mathcal{H} \mid e_i \mid [\,] \mid \exists \overrightarrow{x^?}, [\![a_0]\!]_{(K\ \vec{x})} \rangle \rightsquigarrow^* a_i \wedge e_i \equiv_s a_i\text{)} \\
 & \langle\!\langle \mathcal{H} \mid a_i \mid [\,] \mid \exists \overrightarrow{x^?}, [\![a_0]\!]_{(K\ \vec{x})} \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-done])} \\
 & a_i
\end{array}
$$

    Given $\mathcal{L} \Rightarrow (\overrightarrow{\exists x : [\![\tau]\!]}, [\![a_0]\!]_{(K\ \vec{x})})$, by Theorem 12[p36], we know $a_0 \equiv_s K\ \overrightarrow{x^?}$ for some $\overrightarrow{x^?}$. Together with $e_0 \equiv_s a_0$ and $e_i \equiv_s a_i$, by congruence of $\equiv_s$, we have $\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x^?} \to e_i} \equiv_s a_i$.
  - there is no branch $(K\ \overrightarrow{x^?})$ such that $\mathcal{L} \Rightarrow [\![a_0]\!]_{(K\ \vec{x})}$.

We continue from (†):

$$
\begin{array}{ll}
 & \langle\!\langle \mathcal{H} \mid a_0 \mid (\texttt{match} \bullet \texttt{with } \overrightarrow{K\ \overrightarrow{x^{\tau}} \to e_i}) :: [\,] \mid \emptyset \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-s-save])} \\
 & \dfrac{}{\langle \mathcal{H} \mid e_i \mid (\texttt{match } a \texttt{ with } K\ \overrightarrow{x^{\tau}} \to (\bullet, \mathcal{S}, \mathcal{L})) :: [\,] \mid \begin{array}{l} \mathcal{L}, \overrightarrow{\exists x : [\![\tau]\!]}, \\ [\![a]\!]_{(K\ \overrightarrow{x})} \end{array} \rangle} \\
\rightsquigarrow^* & \text{(By [IH], } \langle \mathcal{H} \mid e_i \mid (\texttt{match } a \texttt{ with } K\ \overrightarrow{x^{\tau}} \to (\bullet, \mathcal{S}, \mathcal{L})) :: [\,] \mid \begin{array}{l} \mathcal{L}, \overrightarrow{\exists x : [\![\tau]\!]}, \\ [\![a]\!]_{(K\ \overrightarrow{x})} \end{array} \rangle \rightsquigarrow^* a_i \\
 & \wedge e_i \equiv_s a_i) \\
 & \dfrac{}{\langle\!\langle \mathcal{H} \mid a_i \mid (\texttt{match } a \texttt{ with } K\ \overrightarrow{x^{\tau}} \to (\bullet, \mathcal{S}, \mathcal{L})) :: [\,] \mid \begin{array}{l} \mathcal{L}, \overrightarrow{\exists x : [\![\tau]\!]}, \\ [\![a]\!]_{(K\ \overrightarrow{x})} \end{array} \rangle\!\rangle} \\
\rightsquigarrow & \text{(By [R-match])} \\
 & \langle\!\langle \mathcal{H} \mid \texttt{match } a_0 \texttt{ with } \overrightarrow{K\ \overrightarrow{x} \to a_i} \mid [\,] \mid \mathcal{L}, \overrightarrow{\exists x : [\![\tau]\!]}, [\![a]\!]_{(K\ \overrightarrow{x})} \rangle\!\rangle \\
\rightsquigarrow & \text{(By [R-done])} \\
 & \texttt{match } a_0 \texttt{ with } \overrightarrow{K\ \overrightarrow{x} \to a_i}
\end{array}
$$

From $e_0 \equiv_s a_0$ and $e_i \equiv_s a_i$, by congruence of $\equiv_s$, we have $\texttt{match } e_0 \texttt{ with } \overrightarrow{K\ \overrightarrow{x} \to e_i} \equiv_s \texttt{match } a_0 \texttt{ with } \overrightarrow{K\ \overrightarrow{x} \to a_i}$.

- Case [S-K]. The proof is similar to the case [S-app]. Simplification of each component e_i to a_i is semantically preserving. After applying induction hypothesis, we apply [R-K]. Given $e_i \equiv_s a_i$, by congruence of $\equiv_s$, $K\ e_1 \ldots e_n \equiv_s K\ a_1 \ldots a_n$.

- Case [S-letL]. We want to show that $(\texttt{let } x = e_1 \texttt{ in } e_2)\ e \equiv_s \texttt{let } x = e_1 \texttt{ in } e_2\ e$. We have:

$$
\begin{array}{ll}
 & (\texttt{let } x = e_1 \texttt{ in } e_2)\ e \\
\to & (\texttt{let } x = val_1 \texttt{ in } e_2)\ e \\
\to & e_2[val_1/x]\ e \\
\to & (\lambda y.a[val_1/x])\ e \\
\to & (\lambda y.a[val_1/x])\ val \\
\to & a[val_1/x, val/y]
\end{array}
$$

and

$$
\begin{array}{ll}
 & \texttt{let } x = e_1 \texttt{ in } e_2\ e \\
\to^* & \texttt{let } x = val_1 \texttt{ in } e_2\ e \\
\to & (e_2\ e)[val_1/x] \\
\to & ((\lambda y.a)\ e)[val_1/x] \\
\to^* & ((\lambda y.a)\ val)[val_1/x] \\
\to^* & a[val/y, val_1/x] \\
= & a[val_1/x, val/y]
\end{array}
$$

By [EqFact], we are done.

- Case [S-matchL]. We want to show that if $fv(e) \cap \overrightarrow{x} = \emptyset$, then ($\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i})\ e \equiv_s \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to (e_i\ e)}$. We have:

$$\begin{array}{rl} & (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i})\ e \\ \to^* & (\texttt{match}\ K\ \overrightarrow{val_x}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e_i})\ e \\ \to & e_i[\overrightarrow{val_x/x}]\ e \\ \to^* & (\lambda y.e_2[\overrightarrow{val_x/x}])\ e \\ \to^* & (\lambda y.e_2[\overrightarrow{val_x/x}])\ val \\ \to & e_2[\overrightarrow{val_x/x}, val/y] \end{array}$$

and

$$\begin{array}{rl} & \texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to (e_i\ e)} \\ \to^* & \texttt{match}\ K\ \overrightarrow{val_x}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to (e_i\ e)} \\ \to & (e_i\ e)[\overrightarrow{val_x/x}] \\ \to^* & (\lambda y.e_2\ e)[\overrightarrow{val_x/x}] \\ \to^* & (\lambda y.e_2\ val)[\overrightarrow{val_x/x}] \\ \to^* & e_2[\overrightarrow{val_x/x}, val/y] \end{array}$$

By [EqFact], we are done.

- Case [S-letR]. We want to show that if $x \notin fv(e)$, then $\lambda y.e\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \equiv_s \texttt{let}\ x = e_1\ \texttt{in}\ \lambda y.e\ e_2$. We have:

$$\begin{array}{rl} & \lambda y.e\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \\ \to^* & \lambda y.e\ (\texttt{let}\ x = val_1\ \texttt{in}\ e_2) \\ \to & \lambda y.e\ (e_2[val_1/x]) \\ \to^* & \lambda y.e\ (val_2[val_1/x]) \\ \to & e[val_2[val_1/x]/y] \\ = & e[val_2/y][val_1/x] \end{array}$$

and

$$\begin{array}{rl} & \texttt{let}\ x = e_1\ \texttt{in}\ (\lambda y.e)\ e_2 \\ \to^* & \texttt{let}\ x = val_1\ \texttt{in}\ (\lambda y.e)\ e_2 \\ \to & ((\lambda y.e)\ e_2)[val_1/x] \\ \to^* & ((\lambda y.e)\ val_2)[val_1/x] \\ \to & e[val_2/y][val_1/x] \end{array}$$

By [EqFact], we are done.

- Case [S-match-match]. We want to show that if $fv(alts) \cap \overrightarrow{x} = \emptyset$, then $\texttt{match}\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})\ \texttt{with}\ alts \equiv_s$ $\texttt{match}\ e_o\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to \texttt{match}\ e\ \texttt{with}\ alts}$. We have:

$$\begin{array}{rl} & \texttt{match}\ (\texttt{match}\ e_0\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})\ \texttt{with}\ alts \\ \to^* & \texttt{match}\ (\texttt{match}\ K\ \overrightarrow{val_0}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to e})\ \texttt{with}\ alts \\ \to & \texttt{match}\ e[\overrightarrow{val_0/x}]\ \texttt{with}\ alts \end{array}$$

and

$$\begin{array}{ll} & \texttt{match}\ e_o\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to \texttt{match}\ e\ \texttt{with}\ alts} \\ \to^* & \texttt{match}\ K\ \overrightarrow{val_0}\ \texttt{with}\ \overrightarrow{K\ \overrightarrow{x} \to \texttt{match}\ e\ \texttt{with}\ alts} \\ \to & (\texttt{match}\ e\ \texttt{with}\ alts)[\overrightarrow{val_0/x}] \\ = & (\text{By } fv(alts) \cap \overrightarrow{x} = \emptyset) \\ & \texttt{match}\ e[\overrightarrow{val_0/x}]\ \texttt{with}\ alts \end{array}$$

By [EqFact], we are done.

- Case [S-match-let]. We want to show if $x \notin fv(alts)$, then $\texttt{match}\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ \texttt{with}\ alts \equiv_s \texttt{let}\ x = e_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts$. We have:

$$\begin{array}{ll} & \texttt{match}\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)\ \texttt{with}\ alts \\ \to^* & \texttt{match}\ (\texttt{let}\ x = val_1\ \texttt{in}\ e_2)\ \texttt{with}\ alts \\ \to & \texttt{match}\ e_2[val_1/x]\ \texttt{with}\ alts \end{array}$$

and

$$\begin{array}{ll} & \texttt{let}\ x = e_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts \\ \to^* & \texttt{let}\ x = val_1\ \texttt{in}\ \texttt{match}\ e_2\ \texttt{with}\ alts \\ \to & (\texttt{match}\ e_2\ \texttt{with}\ alts)[val_1/x] \\ = & (\text{By } x \notin fv(alts)) \\ & \texttt{match}\ e_2[val_1/x]\ \texttt{with}\ alts \end{array}$$

By [EqFact], we are done.

□

Lemma 22 (Correctness of rebuilding)**.** *For all $\mathcal{H}, a_1, \mathcal{S}, \mathcal{L}$, if $\langle\!\langle \mathcal{H} \mid a_1 \mid s :: \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow^* a$, then $a_1 \equiv_s a$.*

Proof. We prove it by induction on the number of transition steps. We have the following induction hypothesis: for all $\mathcal{H}, a_1, \mathcal{S}, \mathcal{L}$, there exists $\mathcal{H}_2, a_2, \mathcal{S}_2, \mathcal{L}_2$, such that $\langle\!\langle \mathcal{H} \mid a_1 \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle \mathcal{H}_2 \mid a_2 \mid \mathcal{S}_2 \mid \mathcal{L}_2 \rangle$ or $\langle\!\langle \mathcal{H} \mid a_1 \mid \mathcal{S} \mid \mathcal{L} \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{H}_2 \mid a_2 \mid \mathcal{S}_2 \mid \mathcal{L}_2 \rangle\!\rangle$,

$$\langle\!\langle \mathcal{H}_2 \mid a_2 \mid \mathcal{S}_2 \mid \mathcal{L}_2 \rangle\!\rangle \rightsquigarrow^* a \wedge a_2 \equiv_s a \quad \text{[IH]}$$

The base case is [R-done]. As two expressions a at both LHS and RHS of $\rightsquigarrow$ are syntactically the same, they are semantically equivalent, so we have the desired result. By [E-exn], [E-ctx], definition of contexts and induction hypothesis [IH], we get the desired result for [R-r-match], [R-r-let], [R-r-fun], [R-r-arg], [R-r-K]. The $\bullet$ in a stack frame indicates the original position of the expression being simplified. It is easy to check that [R-lam], [R-fun], [R-app] and [R-K] just put the simplifed expression back to the $\bullet$ so they are correct. By [E-beta] and [S-var1], [R-beta] is correct. We now consider those slightly non-obvious transitions.

- Case [R-K-match]. This transition implements the simplification rule [K-match] in Figure 11. We want to show that $\texttt{match}\ K\ a_1 \ldots a_n\ \texttt{with}\ \{\ldots; K\ x_1 \ldots x_n \to e; \ldots\} \equiv_s \texttt{let}\ x_1 = a_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = a_n\ \texttt{in}\ e$. We have:

$$\begin{array}{ll} & \texttt{match}\ K\ a_1 \ldots a_n\ \texttt{with}\ \{\ldots; K\ x_1 \ldots x_n \to e; \ldots\} \\ \to^* & \texttt{match}\ K\ val_1 \ldots val_n\ \texttt{with}\ \{\ldots; K\ x_1 \ldots x_n \to e; \ldots\} \\ \to & e[\overrightarrow{val/x}] \end{array}$$

and

$$\begin{array}{ll} & \texttt{let}\ x_1 = a_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = a_n\ \texttt{in}\ e \\ \rightarrow^* & \texttt{let}\ x_1 = val_1\ \texttt{in}\ \ldots\ \texttt{let}\ x_n = val_n\ \texttt{in}\ e \\ \rightarrow & e[\overrightarrow{val/x}] \end{array}$$

By [EqFact], we are done.

- Case [R-s-match]. Given $\mathcal{L} \Rightarrow \exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{K\ \overrightarrow{x}}$ is valid and $a \rightarrow^* K_i\ \overrightarrow{val}$ for some $\overrightarrow{val}$, by Theorem 12[p36], $\mathcal{L} \Rightarrow \exists \overrightarrow{x}, [\![K_i\ \overrightarrow{val}]\!]_{K\ \overrightarrow{x}}$ is valid. From Figure 15, we know $K_i = K$. By [E-match], we get the body e in the branch K. Since $\mathcal{L} \Rightarrow \exists \overrightarrow{x}, [\![a]\!]_{K\ \overrightarrow{x}}$ implies $\mathcal{L} \wedge \exists \overrightarrow{x}, [\![a]\!]_{K\ \overrightarrow{x}}$, [R-s-match] is correct.

- Case [R-s-save]. This transition simplifies each branches with the assumption that $\exists \overrightarrow{x}, [\![a]\!]_{(K\ \overrightarrow{x})}$. Given $\mathcal{L} \wedge \exists \overrightarrow{x : [\![\tau]\!]}, [\![a]\!]_{K\ \overrightarrow{x}}$ is valid and $a \rightarrow^* K_i\ \overrightarrow{val}$ for some $\overrightarrow{val}$, by Theorem 12[p36], $\mathcal{L} \wedge \exists \overrightarrow{x}, [\![K_i\ \overrightarrow{val}]\!]_{K\ \overrightarrow{x}}$ is valid. From Figure 15, we know $K_i = K$. By [E-match], we get the body e in the branch K. So [R-s-save] is correct.

- Case [R-match]. This rule just put back each simplified branch to its original position indicated by the $\bullet$. The $\mathcal{S}$ and $\mathcal{L}$ keep the stack and logical store before each branches are simplified. So [R-match] is correct.

- Case [R-let-save]. The local `let` defines x, by Theorem 11[p36], $\exists x : [\![\tau]\!], [\![a]\!]_x$ is valid. So [R-let-save] is correct.

□

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399