IMP with exceptions over decorated logic

Burak Ekici *

Laboratoire Jean Kuntzmann,
Université Joseph Fourier, Grenoble, France.
Burak.Ekici@imag.fr.

Abstract. In this paper, we separately design *the decorated logic* with respect to the state and the exception effects. Then, we combine two logics to be able to establish small-step semantics of IMP imperative language with exceptional abilities, in a decorated setting. We implement the decorated framework in Coq and certify program equivalence proofs written in that context.

Keywords: Decorated logic, proofs of programs, proof verification, Coq.

1 Introduction

In mostly used imperative programming languages (such as C/C++ and Java), computational effects do exist. With no doubt, they bring an ease and flexibility to the coding process. However, the problem becomes explicit when to prove the properties of programs involving effects. The major difficulty in such kind of a reasoning is the mismatch between the syntax of operations with effects and their interpretation. Typically, a piece of program with arguments in X that returns a value in Y is not interpreted as a function from X to Y due to the effects. The best-known algebraic approach to the problem was initiated by Moggi and implemented in Haskell. There, the main focus is to interpret programs with effects through the monads: the interpretation looks like a function from X to $T(Y)$ where T is a monad. This approach has been extended to Lawvere theories and algebraic handlers [10,11] while there are some others relying on effect systems [8,12] or Hoare Logic [13]. In [6], Duval et al. proposes yet another approach where algebraic theories and effect systems are mixed by adding *decorations* to the terms and equations keeping their interpretations close to syntax in reasoning with effects. In this paper, we first introduce small-step semantics for IMP with exceptional abilities (IMP+Exc). This follows the same approach given in [7]. Then, Duval's decorated logic has been designed for the state and the exception effects, first separately then combined. The combination here means "merging" the behind logics. Next, we establish small-step semantics of IMP+Exc over the combined decorated settings. There, we are able to cope with termination-guaranteed programs. We illustrate the program equivalence proofs within that context and certify proofs with the Coq Proof Assistant.

* the author is a PhD student and the paper itself belongs to the research category.

2 IMP with exceptional abilities

IMP is a standard imperative programming language. It natively provides global variables of type integer, standard integer arithmetic and boolean expressions enriched with a set of commands that is made of do-nothing, assignment, sequence, conditionals and looping operations. Below, we detail the syntax where $\mathtt{n}$ represents a constant integer term while $\mathtt{x}$ is an integer global variable. Note also that abbreviations aexp and bexp respectively denote *arithmetic* and *boolean* expressions as well as cmd stands for *commands*.

$$\begin{aligned}
&\text{aexp: } \mathtt{a_1\ a_2} ::= \mathtt{n} \mid \mathtt{x} \mid \mathtt{a_1} + \mathtt{a_2} \mid \mathtt{a_1} \times \mathtt{a_2}\\
&\text{bexp: } \mathtt{b_1\ b_2} ::= \mathtt{true} \mid \mathtt{false} \mid \mathtt{a_1} = \mathtt{a_2} \mid \mathtt{a_1} \neq \mathtt{a_2} \mid \mathtt{a_1} > \mathtt{a_2} \mid \mathtt{a_1} < \mathtt{a_2} \mid\\
&\qquad\qquad \mathtt{b_1} \wedge \mathtt{b_2} \mid \mathtt{b_1} \vee \mathtt{b_2}\\
&\text{cmd: } \mathtt{c_1\ c_2} ::= \mathtt{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{c_1}; \mathtt{c_2} \mid \mathtt{if\ b\ then\ c_1\ else\ c_2} \mid \mathtt{while\ b\ do\ c_1}
\end{aligned}$$

Fig. 1. Standard IMP syntax

Neither arithmetic nor boolean expressions are allowed to modify the state: they are either pure or read-only. Indeed, small-step semantics for expressions is a total function of the form: $[\![\mathtt{exp}]\!] \times s \to \mathtt{exp}$. It constitutes a new expression out of an input expression and the current state. We recursively define it as follows:

$$\begin{aligned}
[\![\mathtt{n}]\!](\mathtt{s}) &= \mathtt{n}\\
[\![\mathtt{x}]\!](\mathtt{s}) &= \alpha(\mathtt{x})\\
[\![\mathtt{exp_1\ op\ exp_2}]\!](\mathtt{s}) &= [\![\mathtt{exp_1}]\!](\mathtt{s})\ [\![\mathtt{op}]\!]\ [\![\mathtt{exp_2}]\!](\mathtt{s})
\end{aligned}$$

Fig. 2. Small-step semantics for expressions

where $[\![\mathtt{op}]\!]$ stands for natural semantics of any syntactically well-defined arithmetic or boolean operation. For instance, no matter the current state $\mathtt{s}$, the expression $[\![\mathtt{5 + 4}\ ]\!](\mathtt{s})$ will evaluate into 9. Note that constant terms are pure.

$$(\mathtt{skip})\frac{}{\mathtt{s},\ (\mathtt{SKIP}; \mathtt{c}) \rightsquigarrow \mathtt{s}, \mathtt{c}} \quad (\mathtt{sequence})\frac{\mathtt{s}, \mathtt{c_1} \rightsquigarrow \mathtt{s}', \mathtt{c_1}'}{\mathtt{s}, (\mathtt{c_1}; \mathtt{c_2}) \rightsquigarrow \mathtt{s}', (\mathtt{c_1}'; \mathtt{c_2})}$$

$$(\mathtt{assign})\frac{}{\mathtt{s},\ (\mathtt{x} := \mathtt{a}) \rightsquigarrow \mathtt{s}\{x \leftarrow [\![\mathtt{a}]\!](\mathtt{s})\}, \mathtt{SKIP}}$$

$$(\mathtt{cond_1})\frac{[\![\mathtt{b}]\!](\mathtt{s}) = \mathtt{true}}{\mathtt{s},\ (\mathtt{cond\ b\ c_1\ c_2}) \rightsquigarrow \mathtt{s}, \mathtt{c_1}} \quad (\mathtt{cond_2})\frac{[\![\mathtt{b}]\!](\mathtt{s}) = \mathtt{false}}{\mathtt{s},\ (\mathtt{cond\ b\ c_1\ c_2}) \rightsquigarrow \mathtt{s}, \mathtt{c_2}}$$

$$(\mathtt{while_1})\frac{[\![\mathtt{b}]\!](\mathtt{s}) = \mathtt{true}}{\mathtt{s},\ (\mathtt{while\ b\ do\ c}) \rightsquigarrow \mathtt{s}, (\mathtt{c};\ \mathtt{while\ b\ do\ c})}$$

$$(\mathtt{while_2})\frac{[\![\mathtt{b}]\!](\mathtt{s}) = \mathtt{false}}{\mathtt{s},\ (\mathtt{while\ b\ do\ c}) \rightsquigarrow \mathtt{s}, \mathtt{SKIP}}$$

Fig. 3. Small-step semantics for commands

The small-step semantics of commands is also a total function defined by the judgment $\mathtt{s} \times \mathtt{c} \rightsquigarrow \mathtt{s}' \times \mathtt{c}'$. That is to say, in the state $\mathtt{s}$, execution of the command $\mathtt{c}$ will change the state into $\mathtt{s}'$ and it remains to execute $\mathtt{c}'$. Details

can be found in Fig. 3. It remains to note that a command $\mathtt{c}$ at some state $\mathtt{s}$ terminates if there exists a state $\mathtt{s}'$ such that $\mathtt{s},\ \mathtt{c} \rightsquigarrow \mathtt{s}',\ \mathtt{SKIP}$ after a finite number of execution steps. Else if such a state $\mathtt{s}'$ does not exist, the command $\mathtt{c}$ runs forever. Mind also that there is no run-time error since any command apart from $\mathtt{SKIP}$ is allowed to execute at any state $\mathtt{s}$. $\mathtt{SKIP}$ alone is used to indicate the final step of some command set.

2.1 Adding exceptional abilities

Providing exceptional abilities to the standard IMP language is about enriching the command set with $\mathtt{throw}$ and $\mathtt{TRY/CATCH}$ blocks. In addition to the ones in Fig. 1, we also consider following commands:

$$\text{cmd: } \mathtt{c_1\ c_2} ::= \ldots \mid \mathtt{throw\ exc} \mid \mathtt{try\ c_1\ catch\ exc} \Rightarrow \mathtt{c_2}$$

Fig. 4. Syntax for additional commands

where $\mathtt{exc}$ is an exception name of a new type $\mathtt{EName}$. There might be different exception names but $\mathtt{EName}$ is the only type within the context that we introduce in this paper. The small-step semantics for $\mathtt{throw}$ and $\mathtt{TRY/CATCH}$ commands are detailed below:

$$(\mathtt{throw})\ \frac{}{\mathtt{s},\ (\mathtt{throw\ exc}; \mathtt{c}) \rightsquigarrow \mathtt{s},\ \mathtt{throw\ exc}}$$

$$(\mathtt{try_1})\ \frac{}{\mathtt{s}, \mathtt{try\ SKIP\ catch\ exc} \Rightarrow \mathtt{c} \rightsquigarrow \mathtt{s}, \mathtt{SKIP}}$$

$$(\mathtt{try_2})\ \frac{}{\mathtt{s}, \mathtt{try\ (throw\ exc)\ catch\ exc} \Rightarrow \mathtt{c} \rightsquigarrow \mathtt{s}, \mathtt{c}}$$

$$(\mathtt{try_3})\ \frac{\mathtt{exc_1\ exc_2 : EName} \quad \mathtt{exc_1} \neq \mathtt{exc_2}}{\mathtt{s}, \mathtt{try\ (throw\ exc_1)\ catch\ exc_2} \Rightarrow \mathtt{c} \rightsquigarrow \mathtt{s}, \mathtt{throw\ exc_1}}$$

Fig. 5. Small-step semantics for additional commands

Exceptional commands are pure with respect to the state effect: they neither use nor modify the program state. However, they introduce another sort of computational effect: the exception. In prior, we stated that the command $\mathtt{SKIP}$ alone indicates the termination of a program. Now, we extend this by stating: $\mathtt{throw\ exc}$ is also an end but an exceptional end.

Recall that the new language is abbreviated as "IMP+Exc" and the idea is to certify equivalences between programs written in that language. To this extend, Section 3 and Section 4 respectively study decorated logics for the state and the exception which are combined in Section 5. Finally in Section 6, we translate IMP+Exc semantics into decorated settings enriched with an implementation in Coq. There, we give a bunch of examples of equivalent code blocks with certified equivalence proofs. One of the main examples involves a program with an infinite $\mathtt{loop}$ inside the $\mathtt{try}$ block in which an exception is thrown. As soon as the exceptional case is met, the program terminates the loop, recovers the exception and continues with an ordinary execution. We will prove that such a

program has both result and effect equivalence with another program, that is just made of assignments, with respect to the state and the exception.

3 Decorated Logic for the state

Even though it is not syntactically mentioned, the usage/modification of the memory state is allowed in imperative languages. For instance, a `C` function may look up the value of a variable as well as another can modify it. That is an ease in coding but in order to prove correctness of programs with such abilities, one has to revert an explicit usage/manipulation of the state. Therefore, any access to the state is treated as a computational effect: a syntactical term $f : X \to Y$ is not interpreted as $f : X \to Y$ unless it is *pure*. Indeed, a term which reads the program state has instead the interpretation: $f : X \times S \to Y$ while another term which updates the state is interpreted as: $f : X \times S \to Y \times S$ where '$\times$' is the product operator and S is the set of possible states. In [4], we proposed a formal system to prove program properties involving the state, while keeping the memory accesses and manipulations implicit. As in [1], decorated logics for states are obtained from equational logics by classifying terms and equations. Terms are classified as *pure* terms, *accessors* or *modifiers*, which is expressed by adding a *decoration* or superscript, respectively (0), (1) and (2): the decoration of a term (or an equation) characterizes the way it may interact with the state. The decoration (0) is reserved for *pure* terms, while (1) is for *read-only* (*accessor*) and (2) is for *read-write* (*modifier*) terms. Equations are classified as *strong* or *weak* equations, denoted respectively by the symbols $\equiv$ and $\sim$. Weak equation relates only the returned values, while strong equation relates both values and the state effect. Let us start with the descriptions of main features: syntax and rules.

3.1 Syntax and rules

Each type is interpreted as a set. In Fig. 6, $\mathbb{1}$ is the set of singleton while $\mathtt{V_i}$ is the set of values that can be stored in any location `i`. Terms represent functions; they are closed under composition and "pairs", π_1 and π_2 represent the canonical projections with $\langle\,\rangle_{\mathtt{X}} : \mathtt{X} \to \mathbb{1}$ being the canonical empty pair for each type `X`. The basic interface functions are $\mathtt{lookup\ i} : \mathbb{1} \to \mathtt{V_i}$ and $\mathtt{update\ i} : \mathtt{V_i} \to \mathbb{1}$. Fundamentally, `lookup` reads the value stored in a given location while `update` stands to modify it. As mentioned, decorations are used to express the state interaction of a given term. In particular, $\mathtt{id}^{(0)}, \pi_1^{(0)}, \pi_2^{(0)}$ and $\langle\,\rangle^{(0)}$ are pure. $\mathtt{lookup}^{(1)}$ is an accessor while $\mathtt{update}^{(2)}$ is a modifier. The usage of decorations provides a new schema where term signatures are constructed without any occurrence of the state set. So that signatures are kept close to syntax. In addition, decorations give us the flexibility to cope with several interpretations of the state: any proof in decorated settings is valid for different state interpretations.

Syntax :

Types: $\mathtt{t}$	::=	$\mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{t} \times \mathtt{t} \mid \mathbb{1} \mid \mathtt{V_i}_{\ \mathtt{s.t.i} \in \mathtt{Loc}}$
Terms: $\mathtt{f}$	::=	$\mathtt{id} \mid \mathtt{f} \circ \mathtt{f} \mid \langle \mathtt{f}, \mathtt{f} \rangle \mid \pi_1 \mid \pi_2 \mid \langle\ \rangle \mid$
		$\mathtt{lookup\ i} \colon \mathbb{1} \to \mathtt{V_i} \mid \mathtt{update\ i} \colon \mathtt{V_i} \to \mathbb{1}$
Decoration for terms: $(\mathtt{d})$	::=	$(0) \mid (1) \mid (2)$
Equations: $\mathtt{e}$	::=	$\mathtt{f} \equiv \mathtt{f} \mid \mathtt{f} \sim \mathtt{f}$

Fig. 6. Syntax for the state

The intended model is built with respect to the set of states, denoted S, which never appears in the syntax. A pure term $p^{(0)} : X \to Y$ is interpreted as a function $p : X \to Y$, an accessor $a^{(1)} : X \to Y$ as a function $a : X \times S \to Y$ and a modifier $m^{(2)} : X \to Y$ as a function $m : X \times S \to Y \times S$. Obviously, pure terms can be seen as accessors and accessors as modifiers on demand. For instance, this allows term compositions to be directly done without recalling the coKleisli composition. The complete characterization is given in [1].

Rules:

(equiv$_\equiv$), (subs$_\equiv$), (repl$_\equiv$) for all decorations

(equiv$_\sim$), (subs$_\sim$) for all decorations, (repl$_\sim$) only when *replaced term* is pure

(unit$_\sim$) $\dfrac{f^{(2)} \colon X \to \mathbb{1}}{f \sim \langle\ \rangle_X}$ (≡-to-∼) $\dfrac{f^{(2)} \equiv g^{(2)}}{f \sim g}$

(ax$_1$) $\dfrac{}{\mathtt{lookup\ i} \circ \mathtt{update\ i} \sim \mathtt{id_V}}$

(ax$_2$) $\dfrac{\textit{for each pair of locations } (\mathtt{i}, \mathtt{j})\ \mathtt{s.t.}\ \mathtt{i} \neq \mathtt{j}}{\mathtt{lookup\ i} \circ \mathtt{update\ j} \sim \mathtt{lookup\ i} \circ \langle\ \rangle_V}$

(eq$_1$) $\dfrac{f_1^{(d_1)} \sim f_2^{(d_2)}}{f_1 \equiv f_2}$ only when $d_1 \leq 1$ and $d_2 \leq 1$

(eq$_2$) $\dfrac{f_1^{(2)}, f_2^{(2)} \colon X \to Y \quad f_1^{(2)} \sim f_2^{(2)} \quad \langle\ \rangle_Y^{(0)} \circ f_1^{(2)} \equiv \langle\ \rangle_Y^{(0)} \circ f_2^{(2)}}{f_1 \equiv f_2}$

(eq$_3$) $\dfrac{\textit{for each loc. } \mathtt{i}, f_1^{(2)}, f_2^{(2)} \colon X \to \mathbb{1} \quad \mathtt{lookup\ i}^{(1)} \circ f_1^{(2)} \sim \mathtt{lookup\ i}^{(1)} \circ f_2^{(2)}}{f_1 \equiv f_2}$

(pair$_1$) $\dfrac{f_1^{(1)} : X \to Y \quad f_2^{(2)} : X \to Z}{\pi_1 \circ \langle f_1, f_2 \rangle \sim f_1}$ (pair$_2$) $\dfrac{f_1^{(1)} : X \to Y \quad f_2^{(2)} : X \to Z}{\pi_2 \circ \langle f_1, f_2 \rangle \equiv f_2}$

Fig. 7. Rules for the state

As stated in Fig. 7, given syntax is enriched with a set of rules with a special focus on decorations. Strong equations form a congruence while weak equations do not: the replacement rule holds only when the replaced term is pure. However, both are defined to be *equivalence relations* with *reflexivity*, *transitivity* and *symmetry* properties. The fundamental equations for states are provided by the rules (ax$_1$) and (ax$_2$). With (ax$_1$), we have $\mathtt{lookup\ i}^{(1)} \circ \mathtt{update\ i}^{(2)} \sim \mathtt{id_V}^{(0)}$. This means that updating the location $\mathtt{i}$ with a value v and then observing the value of the same location does return v. Clearly this is only a weak equation: its right-hand side does not modify the state while its left-hand side usually does. With (ax$_2$), $\mathtt{lookup\ i}^{(1)} \circ \mathtt{update\ j}^{(2)} \sim \mathtt{lookup\ i}^{(1)} \circ \langle\ \rangle_{\mathtt{V}}{}^{(0)}$, we assume that updating the

location `j` with a value v and then reading the content of the location `i` would return the same result with first forgetting the value v then observing the content of the location `i`. They definitely have different effects on the state. Mind also that this assumption is valid when $\mathtt{i} \neq \mathtt{j}$. There is a free conversion from strong to weak equations ($\equiv$-to-$\sim$). Any term $\mathtt{f} : \mathtt{X} \to \mathbb{1}$ with no result returned (`void`) is said to have an evident result equivalence with the canonical empty pair $\langle\,\rangle_{\mathtt{X}}$ by (unit$_\sim$). In addition strong and weak equations coincide on accessors given by the rule (eq$_1$). Two modifiers $f_1^{(2)}, f_2^{(2)} : X \to Y$ modify the state in the same way if and only if $\langle\,\rangle_Y \circ f_1 \equiv \langle\,\rangle_Y \circ f_2 : X \to \mathbb{1}$, where $\langle\,\rangle_Y : Y \to \mathbb{1}$ throws out the returned value. Then weak and strong equations are related by the property that $f_1 \equiv f_2$ holds if and only if $f_1 \sim f_2$ and $\langle\,\rangle_Y \circ f_1 \equiv \langle\,\rangle_Y \circ f_2$, stated by the rule (eq$_2$). For each location `i`, this can be expressed as a pair of weak equations: $f_1 \sim f_2$ and $\mathtt{lookup\ i} \circ \langle\,\rangle_Y \circ f_1 \sim \mathtt{lookup\ i} \circ \langle\,\rangle_Y \circ f_2$, says the rule (eq$_3$). With (pair$_1$) and (pair$_2$) categorical pairs are characterized: the pair structure $\langle f_1, f_2\rangle$ cannot be used when f_1 and f_2, both are modifiers, since it may lead to a conflict on the returned result. However, it can be used only when f_1 is an accessor. We state by (pair$_1$) that $\langle f_1, f_2\rangle^{(2)}$ has only result equivalence with $f_1^{(1)}$ and by (pair$_2$) that it has both result and effect equivalence with $f_2^{(2)}$.

3.2 Decorated logic for the state in Coq

We represent the set of memory locations by a Coq parameter `Loc: Type`. Since memory locations may contain different types of values, we also assume an arrow type `Val` : `Loc` → `Type` that is the type of values contained in each location.

```
Parameter Loc: Type.  Parameter Val: Loc → Type.
```

Fig. 8. Locations and values in Coq

The terms of the logic are defined through the inductive type named `term` which establishes a new `Type` out of two input `Types`. The type `term Y X` is dependent. It depends on the `Type` instances `X` and `Y` and represents the arrow type: `X` → `Y`. The constructor `tpure` takes a Coq side (pure) function and drops it into the decorated environment. So that pure terms as `id`, π_1, π_2 and $\langle\,\rangle$ are covered within the scope of `tpure`.

```
Inductive term: Type → Type → Type :=
 | comp : ∀ {X Y Z: Type}, term X Y → term Y Z
          → term X Z
 | pair : ∀ {X Y Z: Type}, term X Z → term Y Z
          → term (X×Y) Z
 | tpure : ∀ {X Y: Type}, (X → Y) → term Y X
 | lookup : ∀ i:Loc, term (Val i) unit
 | update : ∀ i:Loc, term unit (Val i).

Infix "o" := comp (at level 60).
```

```
Inductive kind := pure | ro | rw.
Inductive is: kind → ∀ X Y, term X Y → Prop :=
 | is_tpure: ∀ X Y (f: X → Y),
               is pure (@tpure X Y f)
 | is_comp: ∀ k X Y Z (f: term X Y) (g: term Y Z),
               is k f → is k g → is k (f o g)
 | is_pair: ∀ k X Y Z (f: term X Z) (g: term Y Z),
               is k f → is k g → is k (pair f g)
 | is_lookup: ∀ i, is ro (lookup i)
 | is_update: ∀ i, is rw (update i)
 | is_pure_ro: ∀ X Y (f: term X Y),
               is pure f → is ro f
 | is_ro_rw: ∀ X Y (f: term X Y), is ro f → is rw f.
```

Fig. 9. Terms and decorations for the state in Coq

Decorations are enumerated: `pure` (0), `ro` (1) and `rw` (2) and inductively assigned to `terms` via the new type `is`. It builds a proposition out of a `term` and a

decoration. I.e., ∀i : Loc, `is ro (lookup i)` is a `Prop` instance, ensuring that `lookup i` is an accessor. Last two constructors define the decoration hierarchies.

```
Definition id {X: Type} : term X X := tpure id.
Definition pi1 {X Y: Type} : term X (X×Y) := tpure fst.
Definition pi2 {X Y: Type} : term Y (X×Y) := tpure snd.
Definition forget {X} : term unit X := tpure (fun _ ⇒ tt).
Definition constant {X: Type} (v: X): term X unit := tpure (fun _ ⇒ v).
Definition perm {X Y}: term (X×Y) (Y×X) := pair pi2 pi1.
Definition invpi1 {X}: term (X×unit) X := pair id forget.
```

Fig. 10. Some derived terms for the state in Coq

Fig. 10 includes derivation of some terms that we latter use. I.e., ⟨ ⟩ is handled via `tpure` and called `forget`. Besides, we state the rules, in Fig 11, up to weak and strong equalities by defining them in a mutually inductive way: mutuality here is used to enable the constructors with both weak and strong equalities.

```
Reserved Notation "x == y" (at level 80).    Reserved Notation "x ∼ y"  (at level 80).
Inductive strong: ∀ X Y, relation (term X Y) :=
 | subs-repl≡: ∀ X Y Z, Proper (@strong X Y ==> @strong Y Z ==> @strong X Z) comp
 | eq1: ∀ X Y (f g: term X Y), is ro f → is ro g → f ∼ g → f == g
 | eq2: ∀ X Y (f g: term Y X), (forget o f == forget o g) → f ∼ g → f == g
 | eq3: ∀ X (f g: term unit X), (∀ i: Loc, lookup i o f ∼ lookup i o g) → f == g
 | pair2: ∀ X Y' Y (f1: term Y X) (f2: term Y' X), is ro f1 → pi2 o pair f1 f2 == f2
with weak: ∀ X Y, relation (term X Y) :=
 | subs∼: ∀ A B C, Proper (@weak C B ==> @strong B A ==> @weak C A) comp
 | repl∼: ∀ A B C (g: term C B), (is pure g) → Proper (@weak B A ==> @weak C A) (comp g)
 | unit∼: ∀ X (f g: term unit X), f ∼ g
 | ax1: ∀ i, lookup i o update i ∼ id
 | ax2: ∀ i j, i≠j → lookup j o update i ∼ lookup j o forget
 | ≡-to-∼: ∀ X Y (f g: term X Y), f == g → f ∼ g
 | pair1: ∀ X Y' Y (f1: term Y X) (f2: term Y' X), is ro f1 → pi1 o pair f1 f2 ∼ f1
where "x == y" := (strong x y) and "x ∼ y" := (weak x y).
```

Fig. 11. Rules for the state in Coq

Now; we can form the primitive properties of the state structure as in [10] but this time with decorations.

1. `annihilation lookup-update` $\forall$ `i` $\in$ `Loc`, `update` $\mathtt{i}^{(2)} \circ$ `lookup` $\mathtt{i}^{(1)} \equiv$ `id unit`$^{(0)}$
2. `interaction lookup-lookup` $\forall$ `i` $\in$ `Loc`, `lookup` $\mathtt{i}^{(1)} \circ$ `forget (Val i)`$^{(0)} \circ$ `lookup` $\mathtt{i}^{(1)} \equiv$ `lookup` $\mathtt{i}^{(1)}$
3. `interaction update-update` $\forall$ `i` $\in$ `Loc`, `update` $\mathtt{i}^{(2)} \circ$ `pi2`$^{(0)} \circ$ `pair(update i, id (Val i))`$^{(2)} \equiv$ `update` $\mathtt{i}^{(2)} \circ$ `pi2`$^{(0)}$
4. `interaction update-lookup` $\forall$ `i` $\in$ `Loc`, `lookup` $\mathtt{i}^{(1)} \circ$ `update` $\mathtt{i}^{(2)} \sim$ `id (Val i)`$^{(0)}$
5. `commutation lookup-lookup` $\forall$ `i` $\neq$ `j` $\in$ `Loc`, `pair(id (Val i), lookup j)`$^{(1)} \circ$ `lookup` $\mathtt{i}^{(1)} \equiv$ `perm j` $\mathtt{i}^{(0)} \circ$ `pair(id (Val j), lookup i)`$^{(1)} \circ$ `lookup` $\mathtt{j}^{(1)}$
6. `commutation update-update` $\forall$ `i` $\neq$ `j` $\in$ `Loc`, `update` $\mathtt{j}^{(2)} \circ$ `pi2`$^{(0)} \circ$ `pair(update i, id (Val j))`$^{(2)} \equiv$ `update` $\mathtt{i}^{(2)} \circ$ `pi1`$^{(0)} \circ$ `pair(id (Val i), update j)`$^{(2)}$
7. `commutation update-lookup` $\forall$ `i` $\neq$ `j` $\in$ `Loc`, `lookup` $\mathtt{j}^{(1)} \circ$ `update` $\mathtt{i}^{(2)} \equiv$ `pi2`$^{(0)} \circ$ `pair(update i, id (Val j))`$^{(2)} \circ$ `pair(id (Val i), lookup j)`$^{(1)} \circ$ `invpi1`$^{(0)}$

Fig. 12. Primitive properties of the state

Then, we prove such properties within the decorated context and get these proofs certified by Coq. In [3], we detail the implementation as well as the Coq certified proof of `commutation update-lookup`. For the definitions of terms `invpi` and `perm` one can refer back to Fig. 10. The complete Coq library with all certified proofs can be found on https://forge.imag.fr/frs/download.php/649/STATES-0.8.tar.gz.

4 Decorated Logic for the exception

Exception handling is provided by most modern programming languages. It allows to deal with anomalous or exceptional events which require special processing. That brings a flexibility to the coding but in order to prove the correctness of such programs one has to revert an explicit interaction with exceptions. Therefore, any interaction with exceptional cases is treated as a new sort of computational effect: a term $f : X \to Y$ is not interpreted as a function $f : X \to Y$ unless it is pure. Indeed, a term which may raise an exception is instead interpreted as a function $f : X \to Y + E$ and similarly, a term which may catch an exception is interpreted as a function $f : X + E \to Y + E$ where '+' is disjoint union operator and E is the set of exceptions. Moreover, it has been shown in [2] that the core part of this proof system is dual to one for the state which is explained in Section 3. As in [3], decorated logics for exception are obtained from equational logics by classifying terms and equations. Terms are classified as *pure* terms, *propagators* or *catchers*, which is expressed by adding a *decoration* or superscript, respectively (0), (1) and (2): the decoration of a term (or an equation) characterizes the way it may cope with exceptional cases. The decoration (0) is reserved for terms which are *pure*, while (1) is for *throwers* and (2) is for *catchers*. Equations are classified as *strong* or *weak* equations, denoted respectively by the symbols $\equiv$ and $\sim$. Weak equation relates the ordinary cases in programs, while strong equations relates both ordinary and exceptional cases. Let us describe the main features of the logic: syntax and rules.

4.1 Syntax and rules

The full syntax is declared in Fig. 13 where $\mathbb{0}$ is the empty type while $\mathtt{V_e}$ represents the set of values which can be used as arguments for the exceptions with name `e`. Terms represent functions; they are closed under composition and "copairs" (or case distinction), `inl` and `inr` represent the canonical inclusions into a coproduct (or disjoint union). The basic functions for dealing with exceptions are `tag e`: $\mathtt{V_e} \to \mathbb{0}$ and `untag e`: $\mathbb{0} \to \mathtt{V_e}$. A fundamental feature of the mechanism of exceptions is the distinction between *ordinary* (or *non-exceptional*) values and *exceptions*. While `tag e` encapsulates its argument (which is an ordinary value) into an exception, `untag e` is applied to an exception for recovering this argument. The usual `throw` and `try/catch` constructions are built from the more basic `tag e` and `untag e` operations [3]. The term `downcast` takes an input term `f` and behaves exactly as `f` on ordinary arguments, if the argument is exceptional then it enforces `f` to propagate it (in case `f` might catch it). As mentioned, we use *decorations* on terms for expressing how they interact with the exceptions. In particular, $\mathtt{id}^{(0)}$, $\mathtt{inl}^{(0)}$, $\mathtt{inr}^{(0)}$ and $[\ ]^{(0)}$ are pure. Clearly $\mathtt{tag\ e}^{(1)}$ and $\mathtt{downcast}^{(1)}$ are throwers while $\mathtt{untag\ e}^{(2)}$ is a catcher. A thrower may throw exceptions and must propagate any given exception, while a catcher may recover from exceptions. Using decorations provides a new schema where term signatures are constructed without any occurrence of a "type of exceptions". Thus, signatures are kept close to the syntax. In addition, decorating

terms gives us the flexibility to cope with more than one interpretation of the exceptions. This means that with such an approach, any proof in decorated logic is valid for different implementations of the exceptions.

Syntax :

Types: $\mathtt{t}$	$::= \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{t} + \mathtt{t} \mid \mathbb{0} \mid \mathtt{V_e}$ s.t.e∈EName
Terms: $\mathtt{f}$	$::= \mathtt{id} \mid \mathtt{f} \circ \mathtt{f} \mid [\mathtt{f} \mid \mathtt{f}] \mid \mathtt{inl} \mid \mathtt{inr} \mid [\,] \mid$
	$\mathtt{tag\ e} : \mathtt{V_e} \to \mathbb{0} \mid \mathtt{untag\ e} : \mathbb{0} \to \mathtt{V_e} \mid \mathtt{downcast}$
Decoration for terms: $(\mathtt{d})$	$::= (0) \mid (1) \mid (2)$
Equations: $\mathtt{e}$	$::= \mathtt{f} \equiv \mathtt{f} \mid \mathtt{f} \sim \mathtt{f}$

Fig. 13. Syntax for the exception

The intended model is built with respect to the set of exceptions, denoted E, which never appears in the syntax. It interprets each type X as a set X, each pure term $u^{(0)} : X \to Y$ as a function $u : X \to Y$, each propagator $a^{(1)} : X \to Y$ as a function $a : X \to Y + E$ and each catcher $f^{(2)} : X \to Y$ as a function $f : X + E \to Y + E$. The complete characterization is given in [3].

Rules:

(equiv$_\equiv$), (subs$_\equiv$), (repl$_\equiv$) for all decorations

(equiv$_\sim$), (repl$_\sim$) for all decorations, (subs$_\sim$) only when *substituted term* is pure

(empty$_\sim$) $\dfrac{f^{(2)} : \mathbb{0} \to X}{f \sim [\,]_X}$ (≡-to-∼) $\dfrac{f^{(2)} \equiv g^{(2)}}{f \sim g}$ (downcast$_\sim$) $\dfrac{f^{(2)} : Y \to X}{\mathtt{downcast}\ f \sim f}$

(eax$_1$) $\dfrac{}{\mathtt{untag\ e} \circ \mathtt{tag\ e} \sim \mathtt{id_V}}$

(eax$_2$) $\dfrac{\textit{for each pair of exception names } (\mathtt{e_1}, \mathtt{e_2}) \text{ s.t. } \mathtt{e_1} \neq \mathtt{e_2}}{\mathtt{untag\ e_1} \circ \mathtt{tag\ e_2} \sim [\,]_V \circ \mathtt{tag\ e_2}}$

(eeq$_1$) $\dfrac{f_1^{(d_1)} \sim f_2^{(d_2)}}{f_1 \equiv f_2}$ only when $d_1 \leq 1$ and $d_2 \leq 1$

(eeq$_2$) $\dfrac{f_1^{(2)}, f_2^{(2)} : Y \to X \quad f_1^{(2)} \sim f_2^{(2)} \quad f_1^{(2)} \circ [\,]_Y^{(0)} \equiv f_2^{(2)} \circ [\,]_Y^{(0)}}{f_1 \equiv f_2}$

(eeq$_3$) $\dfrac{\textit{for exc. name } \mathtt{e}, f_1^{(2)}, f_2^{(2)} : \mathbb{0} \to X \quad f_1^{(2)} \circ \mathtt{tag\ e}^{(1)} \sim \circ f_2^{(2)} \circ \mathtt{tag\ e}^{(1)}}{f_1 \equiv f_2}$

(copair$_1$) $\dfrac{f_1^{(1)} : X \to Y \quad f_2^{(2)} : Z \to Y}{[f_1 \mid f_2] \circ \mathtt{inl} \sim f_1}$ (copair$_2$) $\dfrac{f_1^{(1)} : X \to Y \quad f_2^{(2)} : Z \to Y}{[f_1 \mid f_2] \circ \mathtt{inr} \equiv f_2}$

Fig. 14. Rules for the exception

As stated in Fig. 14, a set of rules enriches the given syntax with a special focus on decorations. Strong equations form a congruence while weak equations do not: the replacement rule holds only when the replaced term is pure. However, both are defined to be *equivalence relations*. Since, `(downcast f)` and `f` behave the same on ordinary values, they are weakly equal: ensured by the (donwcast$_\sim$) rule. The fundamental equations for states are provided by the rules (eax$_1$) and (eax$_2$). With (ax$_1$), we have $\mathtt{untag\ e}^{(2)} \circ \mathtt{tag\ e}^{(1)} \sim \mathtt{id_V}^{(0)}$. This means that encapsulating the argument with an exception of name `e` followed by an immediate recovery would be equivalent to "doing nothing" considering the ordinary arguments. Clearly this is only a weak equation: its right

side has nothing to do with exceptional cases while left side has. With (eax_2), $\texttt{untag}\ \texttt{e}_1{}^{(2)} \circ \texttt{tag}\ \texttt{e}_2{}^{(1)} \sim [\,]_V \circ \texttt{tag}\ \texttt{e}_2{}^{(1)}$, we assume on the left that encapsulating an argument with an exception of name $\texttt{e}_2$ and then recovering from a different exception of name $\texttt{e}_1$ would just lead $\texttt{e}_2$ to be propagated. Whilst on the right, the argument is encapsulated with $\texttt{e}_2$ with no recovery attempt afterwards. Thus, they behave different on exceptional arguments but the same on ordinary ones so that the equality in between is weak. There is a free conversion from strong to weak equations ($\equiv$-to-$\sim$). Any term $\texttt{f} : \mathbb{0} \to \texttt{X}$ with no input parameter is said to have an equivalence on ordinary arguments with the empty copair $[\,]_{\texttt{X}}$ by ($\text{empty}_\sim$). In addition, strong and weak equations coincide on propagators given by the rule (eeq_1). Two catchers $f_1^{(2)}, f_2^{(2)} : X \to Y$ have the same effect with respect to the exceptional arguments if and only if $f_1 \circ [\,]_Y \equiv f_2 \circ [\,]_Y : \mathbb{0} \to X$. Then weak and strong equations are related by the property that $f_1 \equiv f_2$ if and only if $f_1 \sim f_2$ and $f_1 \circ [\,]_Y \equiv f_2 \circ [\,]_Y$, by the rule ($\text{eeq}_2$). For each exception name $\texttt{e}$, this can be expressed as a pair of weak equations: $f_1 \sim f_2$ and $f_1 \circ [\,]_Y \circ \texttt{tag e} \sim f_2 \circ [\,]_Y \circ \texttt{tag e}$, ensured by the rule ($\text{eeq}_3$). With ($\text{copair}_1$) and ($\text{copair}_2$), categorical copairs are characterized: the copair structure $[f_1 \mid f_2]$ cannot be used when both f_1 and f_2 are catchers, since it may lead to conflicts on exceptional arguments. However, it can be used only when f_1 is a propagator. We state with (copair_1) that ordinary arguments are treated by $[f_1 \mid f_2]^{(2)}$ as they would be by $f_1^{(1)}$ and with (copair_2) that ordinary and exceptional arguments are treated by $[f_1 \mid f_2]^{(2)}$ as they would be by $f_2^{(2)}$.

4.2 Decorated logic for the exception in Coq

Coq implementation follows the same approach with the one for the state. We represent the set of exception names by a Coq parameter `EName: Type`. We also assume an arrow type `Val : EName → Type` which is the set of parameters for each exception name. Then, we inductively define terms and assign decorations as in Fig.15. Note that in order to indicate the decorations, we respectively use keywords `pure`, `propagator` and `thrower` instead of (0), (1) and (2).

```
Parameter EName: Type.  Parameter EVal: EName → Type.

Inductive term: Type → Type → Type :=
 | downcast : ∀{X Y} (f: term Y X), term Y X
 | copair : ∀ {X Y Z: Type}, term Z X → term Z Y
          → term Z (X + Y)
 | tpure : ∀ {X Y: Type}, (X → Y) → term Y X
 | tag : ∀ e:EName, term Empty_set (EVal e)
 | untag : ∀ e:EName, term (EVal e) Empty_set.

Inductive kind := pure | propagator | catcher.

Inductive is: kind → ∀ X Y, term X Y → Prop :=
 | is_downcast: ∀ X Y (f: term Y X),
                is propagator (downcast f)
 | is_tpure: ∀ X Y (f: X → Y),
                is pure (@tpure X Y f)
 | is_copair: ∀ k X Y Z (f: term X Z)
    (g: term Y Z), is k f → is k g → is k (pair f g)
 | is_tag: ∀ i, is propagator (tag e)
 | is_untag: ∀ i, is catcher (untag e)
 | is_pure_propagator: ∀ X Y (f: term X Y),
                is pure f → is propagator f
 | is_propagator_catcher: ∀ X Y (f: term X Y),
                is propagator f → is catcher f.
```

Fig. 15. Terms and decorations for the exception in Coq

```
Definition id {X: Type} : term X X := tpure id.
Definition emptyfun (X: Type) (e: Empty_set) : X := match e with end.
Definition empty X: term X Empty_set := tpure (emptyfun X).
Definition inl {X Y} : term (X+Y) X := tpure inl.
Definition inr {X Y} : term (X+Y) Y := tpure inr.
Definition throw (X: Type) (e: EName): term X unit := (empty X) o tag e.
Definition TRY_CATCH (X Y: Type) (e:EName) (f: term Y X) (g: term Y unit)
  := downcast(copair (@id Y) (g o untag e) o inl o f).
Definition ttrue : term (unit+unit) unit := inl.
Definition ffalse : term (unit+unit) unit := inr.
```

Fig. 16. Some derived terms for the exception in Coq

Some derived terms including `throw` and `TRY/CATCH` blocks are given in Fig.16. The functions `inl` and `inr` indicate coprojections. In addition, $[\,]$ is called `empty`. `ttrue` and `ffalse` correspond to boolean *true* and *false*. The operation `throw e` is just `tag`ing an exception of name `e` followed by $[\,]_X$ which is used to bridge the execution to the next command. Within the scope of the intended model, it is used to include $\mathbb{0}$ into $\mathbb{0}+X$. To build (`TRY f CATCH e g`), we use copairs to have case distinction: (1) either the term `f` does not throw an exception so that the term `g` is never triggered. That corresponds to the $\mathtt{id}_Y$ case of the copair. (2) or else, the code `f` throws an exception then through `untag e`, the exception would be recovered (if pattern matching is fine with exception names) and execution continues with the term `g`. The whole $\mathtt{TRY}-\mathtt{CATCH}$ block is either pure (in case no exceptional case has been met) or a thrower/propagator (in case, the thrown exception by `f` has not been caught or a previously thrown exception has been propagated). This is ensured the rule (downcast$_\sim$). Now; we get the rules in Coq:

```
Reserved Notation "x == y" (at level 80).    Reserved Notation "x ~ y"  (at level 80).
Definition pure_id X Y (x y: term X Y) := is pure x ∧ x = y.

Inductive strong: ∀ X Y, relation (term X Y) :=
  | subs-repl≡: ∀ X Y Z, Proper (@strong X Y ==> @strong Y Z ==> @strong X Z) comp
  | eeq1: ∀ X Y (f g: term X Y), is propagator f → is propagator g → f ~ g → f == g
  | eeq2: ∀ X Y (f g: term X Y), (f o empty == g o empty) → f ~ g → f == g
  | eeq3: ∀ X (f g: term X Empty_set), (∀ e: EName, f o tag e ~ g o tag e) → f == g
  | copair2: ∀ X Y Z (f1: term Y X) (f2: term Y Z), is propagator f1 → pair f1 f2 o inr == f2
  | s-equiv1: ∀ X Y (f: term X Y), f == f
with weak: ∀ X Y, relation (term X Y) :=
  | subs~ : ∀ A B C, Proper (@weak C B ==> @pure_id B A ==> @weak C A) comp
  | repl~ : ∀ A B C, Proper (@strong C B ==> @weak B A ==> @weak C A) comp
  | empty~: ∀ X (f g: term X Empty_set), f ~ g
  | donwcast~: ∀ X Y (f: term Y X), downcast f ~ f
  | eax1: ∀ e, untag e o tag e ~ id
  | eax2: ∀ e1 e2, e1≠e2 → untag e1 o tag e2 ~ empty o tag e2
  | ≡-to-~: ∀ X Y (f g: term X Y), f == g → f ~ g
  | copair1: ∀ X Y Z (f1: term Y X) (f2: term Y Y), is propagator f1 → copair f1 f2 o inl ~ f1
where "x == y" := (strong x y) and "x ~ y" := (weak x y).
```

Fig. 17. Rules for the exception in Coq

1. `propagator propagates:` $\forall \mathtt{g}^{(1)} : \mathtt{Y} \to \mathtt{X},\ \mathtt{g}^{(1)} \circ [\,]_{\mathtt{Y}}^{(0)} \equiv [\,]_{\mathtt{X}}^{(0)}$
2. `annihilation untag-tag:` $\mathtt{tag\ t}^{(1)} \circ \mathtt{untag\ t}^{(2)} \equiv \mathtt{id}\ \mathbb{0}^{(0)}$
3. `annihilation catch-raise:` $(\mathtt{TRY} - \mathtt{CATCH\ f\ (t} \Rightarrow \mathtt{(throw\ t\ Y)))}^{(1)} \equiv \mathtt{f}^{(1)}$
4. `commutation untag-untag:` $\mathtt{s} \neq \mathtt{t},\ (\mathtt{untag\ t}^{(2)} + \mathtt{id\ s}^{(0)}) \circ \mathtt{untag\ s}^{(2)} \equiv (\mathtt{id\ t}^{(0)} + \mathtt{untag\ s}^{(2)}) \circ \mathtt{untag\ t}^{(2)}$
5. `interaction propagator-throw:` $\mathtt{g}^{(1)} : \mathtt{Y} \to \mathtt{X},\ \mathtt{g}^{(1)} \circ \mathtt{(throw\ t\ Y)} \equiv \mathtt{(throw\ t\ X)}$
6. `commutation catch-catch:` $\mathtt{s} \neq \mathtt{t}, (\mathtt{TRY} - \mathtt{CATCH\ f\ (t} \Rightarrow \mathtt{g} \mid \mathtt{s} \Rightarrow \mathtt{h}))^{(1)} \equiv (\mathtt{TRY} - \mathtt{CATCH\ f\ (s} \Rightarrow \mathtt{h} \mid \mathtt{t} \Rightarrow \mathtt{g}))^{(1)}$

Fig. 18. Primitive properties of the exception

After all, we give the properties of the exception followed by the related proofs certified in Coq. In [3], we detail the implementation and the certified proof of

the `propagator propagates`. The complete Coq library with all certified proofs is available on https://forge.imag.fr/frs/download.php/648/EXCEPTIONS-0.3.tar.gz.

5 Combination: the state & the exception

In order to formally cope with both the state and the exception effects in the same program, one needs to combine the related formal models. For instance in Haskell, effects are modeled by monads and combination is done through monad transformers. However, here we just merge the related decorated logics. Let us start with explanation of the syntax:

`Syntax` :		
Types: $\mathtt{t}$	$::=$	$\mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{t} + \mathtt{t} \mid \mathtt{t} \times \mathtt{t} \mid \mathbb{1} \mid \mathbb{0} \mid$
		$\mathtt{V_i}_{\ \mathtt{s.t.i} \in \mathtt{Loc}} \mid \mathtt{V_e}_{\ \mathtt{s.t.e} \in \mathtt{EName}}$
Terms: $\mathtt{f}$	$::=$	$\mathtt{id} \mid \mathtt{f} \circ \mathtt{f} \mid [\mathtt{f} \mid \mathtt{f}] \mid \langle \mathtt{f}, \mathtt{f} \rangle \mid$
		$\mid [\,] \mid \langle\,\rangle \mid \mathtt{inl} \mid \mathtt{inr} \mid \pi_1 \mid \pi_2 \mid \mathtt{downcast} \mid$
		$\mathtt{lookup\ i} \colon \mathbb{1} \to \mathtt{V_i} \mid \mathtt{update\ i} \colon \mathtt{V_i} \to \mathbb{1} \mid$
		$\mathtt{tag\ e} \colon \mathtt{V_e} \to \mathbb{0} \mid \mathtt{untag\ e} \colon \mathbb{0} \to \mathtt{V_e}$
Decoration for terms: $(\mathtt{d})$	$::=$	$(0,0) \mid (0,1) \mid (0,2) \mid (1,0) \mid (1,1) \mid (1,2) \mid$
		$(2,0) \mid (2,1) \mid (2,2)$
Equations: $\mathtt{e}$	$::=$	$\mathtt{f} \equiv\!\equiv \mathtt{f} \mid \mathtt{f} \equiv\!\sim \mathtt{f} \mid \mathtt{f} \sim\!\equiv \mathtt{f} \mid \mathtt{f} \sim\!\sim \mathtt{f}$

Fig. 19. Syntax for the combined state and exception

Types and terms are simply unionized. The decorations are paired off to cover all possible combinations: left component is given with respect to the state while right is to the exception. I.e., $f^{(1,2)}$ says that f may *access* to the state alongside *catching* exceptions. The hierarchies among decorations are preserved: $\frac{f^{(0,d)}}{f^{(1,d)}}$, $\frac{f^{(1,d)}}{f^{(2,d)}}$, $\frac{f^{(d,0)}}{f^{(d,1)}}$ and $\frac{f^{(d,1)}}{f^{(d,2)}}$. Obviously, we have all possible combinations of equalities with preserved hierarchies: $(\equiv\equiv\text{-to-}\equiv\sim)\frac{f \equiv\equiv g}{f \equiv\sim g}$, $(\equiv\equiv\text{-to-}\sim\equiv)\frac{f \equiv\equiv g}{f \sim\equiv g}$ and $(\sim\sim\text{-to-}\equiv\equiv)\frac{f^{(d_1,d_2)} \sim\sim\ g^{(d_3,d_4)}}{f \equiv\equiv\ g}$ only when $d_1, d_2, d_3, d_4 \leq 1$. Here we form the combined rules:

1. $\equiv\equiv$ relates the properties that are strongly equal both with respect to the state and the exception: (eq_1) is now with $f_1^{(d_1,2)}$, $f_2^{(d_2,2)}$, (eq_2) and (eq_3) with $f_1^{(2,2)}$, $f_2^{(2,2)}$ and (pair_2) with $f_1^{(1,2)}$, $f_2^{(2,2)}$. In addition, (eeq_1) with $f_1^{(2,d_1)}$, $f_2^{(2,d_2)}$, (eeq_2) and (eeq_3) with $f_1^{(2,2)}$, $f_2^{(2,2)}$ and (copair_2) with $f_1^{(2,1)}$, $f_2^{(2,2)}$.
2. $\sim\equiv$ relates the properties that are weakly equal with respect to the state: $(\text{unit}_\sim)$ is now with $f^{(2,2)}$, (ax_1) and (ax_2) with $\mathtt{lookup}^{(1,0)}$, $\mathtt{update}^{(2,0)}$ and (pair_1) with $f_1^{(1,2)}$, $f_2^{(2,2)}$
3. $\equiv\sim$ relates the properties that are weakly equal with respect to the exception: $(\text{empty}_\sim)$ is with $f^{(2,2)}$, $(\text{downcast}_\sim)$ with $f^{(2,2)}$, (eax_1) and (eax_2) with $\mathtt{tag}^{(0,1)}$, $\mathtt{untag}^{(0,2)}$ and (copair_1) with $f_1^{(2,1)}$, $f_2^{(2,2)}$
4. $\sim\sim$ relates nothing but the conversions: $\sim\equiv$ and $\equiv\sim$ can be seen as $\sim\sim$.

6 IMP+Exc over decorated logic

Finally, it comes to translate the semantics detailed in Section 2 into the combined decorated settings. Given that IMP only provides the integer data type, the values that can be stored in any location `i` are just integers. So that any occurrence of (`Val i`) in term signatures is replaced by $\mathbb{Z}$. Here, we start with expressions and recursively define the translator function `dExp`. It mainly takes an expression and outputs a decorated term of type `term` $\mathbb{Z}$ `unit` or `term` $\mathbb{B}$ `unit` depending on the input expression type. Below, we have it recursively defined:

$$\begin{array}{rcl}
\texttt{dExp n} & \Rightarrow & (\texttt{constant n})^{(0,0)} \\
\texttt{dExp x} & \Rightarrow & (\texttt{lookup x})^{(1,0)} \\
\texttt{dExp (f exp)} & \Rightarrow & (\texttt{tpure f})^{(0)} \circ (\texttt{dExp exp})^{(1,0)} \\
\texttt{dExp } \langle \texttt{exp}_1, \texttt{exp}_2 \rangle & \Rightarrow & \langle \texttt{dExp exp}_1, \texttt{dExp exp}_2 \rangle^{(1,0)}
\end{array}$$

Fig. 20. Translating expressions into decorated settings

where `f` is a unary pure term. Besides, we have some additional rules to make use of some pure algebraic operations in the decorated setting. Before going into the rule details, we define some terms that help to form them: given in Fig. 21, `lpi` is the syntactical term providing loop iteration(s) together with the rule (`imp-loopiter`) while `pbl` forms terms of type `term` (`unit` + `unit`) $\mathbb{B}$ for compatibility issues in rule statements ($\texttt{imp}_2$) and ($\texttt{imp}_4$).

$$\begin{array}{rcl}
\texttt{lpi } (\texttt{b} : \texttt{term unit } (\texttt{unit} + \texttt{unit})) \ (\texttt{f} : \texttt{term unit unit}) & := & \texttt{tpure } (\lambda \texttt{x} : \texttt{unit.x}). \\
\texttt{pbl} & := & \texttt{tpure } (\texttt{bool_to_two}) \\
\text{where } \texttt{bool_to_two } (\texttt{b} : \texttt{bool}) & := & (\texttt{if b then } (\texttt{inl tt}) \texttt{ else } (\texttt{inr tt})). \\
\text{such that } \texttt{tt} : \texttt{unit} \text{ and } \texttt{inl}, \ \texttt{inr} : \texttt{unit} \to (\texttt{unit} + \texttt{unit})
\end{array}$$

Fig. 21. Additional terms: IMP specific

$$(\texttt{imp} - \texttt{loopiter}) \ \frac{\forall (\texttt{b} : \texttt{term unit } (\texttt{unit} + \texttt{unit})) \ (\texttt{f} : \texttt{term unit unit})}{\texttt{lpi b f} \equiv\equiv \left[(\texttt{lpi b f}) \circ \texttt{f} \middle| \texttt{id} \right] \circ \texttt{b}}$$

$$(\texttt{imp}_1) \ \frac{\forall \texttt{p}, \ \texttt{q} : \mathbb{Z}, \ (\texttt{f} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z})}{\texttt{tpure f} \circ \langle \texttt{constant p}, \texttt{constant q} \rangle \equiv\equiv (\texttt{constant f}(\texttt{p}, \texttt{q}))}$$

$$(\texttt{imp}_2) \ \frac{\forall \texttt{p}, \ \texttt{q} : \mathbb{Z}, \ (\texttt{f} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}) \ \ \texttt{f}(\texttt{p}, \texttt{q}) = \texttt{false}}{\texttt{pbl} \circ \texttt{tpure f} \circ \langle \texttt{constant p}, \texttt{constant q} \rangle \equiv\equiv \texttt{ffalse}}$$

$$(\texttt{imp}_4) \ \frac{\forall \texttt{p}, \ \texttt{q} : \mathbb{B}, \ (\texttt{f} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}) \ \ \texttt{f}(\texttt{p}, \texttt{q}) = \texttt{false}}{\texttt{pbl} \circ \texttt{tpure f} \circ \langle \texttt{constant p}, \texttt{constant q} \rangle \equiv\equiv \texttt{ffalse}}$$

$$(\texttt{imp}_6) \frac{\texttt{f} : \texttt{Y} \to \texttt{Z} \ \ \texttt{g} : \texttt{X} \to \texttt{Y}}{\texttt{tpure f} \circ \texttt{tpure g} \equiv\equiv \texttt{tpure } (\lambda \texttt{x.f}(\texttt{g x}))}$$

$$(\texttt{imp}_7) \frac{\texttt{f g} : \texttt{Y} \to \texttt{X} \ \ (\forall \texttt{x}, \ \texttt{f x} \ = \ \texttt{g x})}{\texttt{tpure f} \equiv \texttt{tpure g}}$$

Fig. 22. Additional rules: IMP specific

In ($\texttt{imp}_2$) and ($\texttt{imp}_4$) by replacing `false` into `true` and `ffalse` into `ttrue` we get ($\texttt{imp}_3$) and ($\texttt{imp}_5$) that are not explicitly stated here. The fact that IMP commands are of type $\mathbb{1} \to \mathbb{1}$, they will be designed in such a way that domains and codomains being set to `unit` within the decorated scope. Now; we recursively define the translator function `dCmd` which establishes a decorated term of type `term unit unit`, out of an input command:

$$
\begin{array}{rcl}
\texttt{dCmd SKIP} & \Rightarrow & (\texttt{id unit})^{(0,0)} \\
\texttt{dCmd (x := a)} & \Rightarrow & (\texttt{update x})^{(2,0)} \circ\ (\texttt{dExp a})^{(1,0)} \\
\texttt{dCmd } (\texttt{c}_1\texttt{; c}_2) & \Rightarrow & (\texttt{dCmd c}_2)^{(2,0)} \circ (\texttt{dCmd c}_1)^{(2,0)} \\
\texttt{dCmd (cond b c}_1\ \texttt{c}_2) & \Rightarrow & \Big[\texttt{dCmd c}_1 \ \Big|\ \texttt{dCmd c}_2\Big]^{(2,0)} \circ\ \texttt{pbl}^{(0,0)} \circ (\texttt{dExp b})^{(1,0)} \\
\texttt{dCmd (while b do c)} & \Rightarrow & \Big[(\texttt{lpi } (\texttt{pbl} \circ (\texttt{dExp b}))\ (\texttt{dCmd c})) \circ (\texttt{dCmd c}) \ \Big|\ \texttt{id}\Big]^{(2,0)} \\
 & & \circ\ \texttt{pbl}^{(0,0)} \circ (\texttt{dExp b})^{(1,0)} \\
\texttt{dCmd (throw exp)} & \Rightarrow & [\,]_{\mathbb{1}}{}^{(0,0)} \circ \texttt{tag exp}^{(0,1)} \\
\texttt{dCmd (try c}_1\ \texttt{catch exp} \Rightarrow \texttt{c}_2) & \Rightarrow & \downarrow (\Big[\texttt{id} \ \Big|\ \texttt{c}_2 \circ \texttt{untag exp}\Big] \circ \texttt{inl} \circ \texttt{c}_1)^{(0,1)}
\end{array}
$$

Fig. 23. Translating commands into decorated settings

Let us take a closer look into conditionals and loops in terms of diagrams:

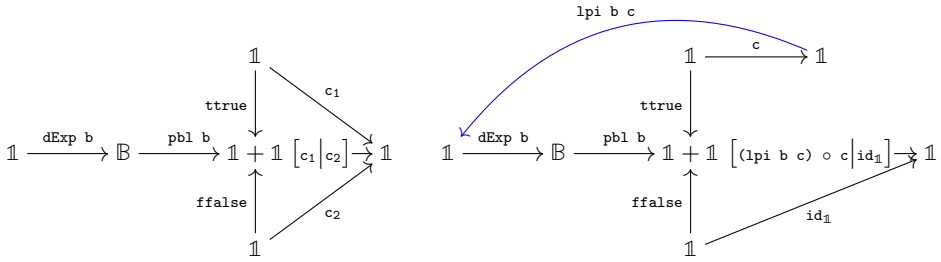


Fig. 24. ($\texttt{cond b c}_1\ \texttt{c}_2$) and (`while b do c`) in decorated settings

there, we use categorical copairs to have case distinction. For instance, in Fig. 24 on the left, after the condition check if the boolean evaluates into `ttrue`, then we have $\texttt{c}_1$ in execution or else $\texttt{c}_2$. The only difference on the right is that as long as the boolean evaluates into `ttrue`, `c` is in execution: diagrammatically, it says that the arrow `lpd b c` is each time replaced by the whole diagram itself. As mentioned, this property is provided by the syntactic term `lpd` and the attached rule `(imp-loopiter)`. When the boolean evaluates into `ffalse`, we have `id` forcing the loop to terminate.

Contrarily, in the translation of `throw` and `try/catch`, the basis is the core decorated operations for the exception effect. Recall that they are defined as they are

given in Section 4.2 with a single difference in the signatures: domains/codomains are now set to $\mathbb{1}$. Below, we have the translation in terms of diagrams:

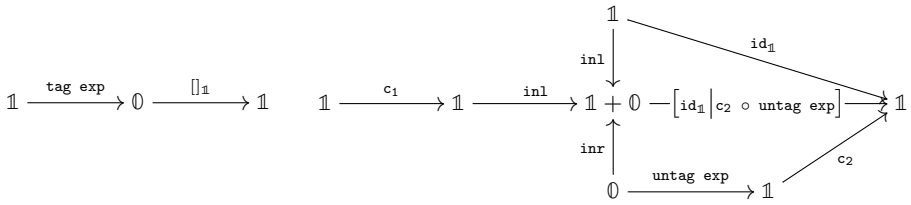


Fig. 25. `(throw exp)` and $(\texttt{try } c_1 \texttt{ catch exp} \Rightarrow c_2)$ in decorated settings

We implement such formalizations in Coq:

```
Inductive Exp : Type → Type :=
 | const : ∀ A, A → Exp A
 | loc : Loc → Exp Z
 | apply : ∀ A B, (A → B) → Exp A → Exp B
 | pExp : ∀ A B, Exp A → Exp B → Exp (A × B).
```

```
Fixpoint dExp A (e: Exp A): term A unit :=
 match e with
  | const Z n ⇒ constant n
  | loc x ⇒ lookup x
  | apply _ _ f x ⇒ tpure f o (dExp x)
  | pExp _ _ x y ⇒ pair (dExp x) (dExp y)
 end.
```

Fig. 26. IMP+Exc expressions in Coq

Expressions are inductively defined forming a new Coq `Type`, `Exp`. Indeed, `Exp` is a dependent type. That means that the type of `Exp A` depends on the term `A: Type`. For instance, when `A` := $\mathbb{B}$, we build the type for boolean expressions while the case `A` := $\mathbb{Z}$ enables us to construct the type for arithmetic expressions. Obviously, `Exp` is polymorphic, too. Speaking of the constructors: an expression might be a constant term (constructed by `const`), a variable (by `loc`), an expression with an applied pure term (by `apply`) or a pair of expressions (by `pExp`). The translation given in Fig. 20 is characterized by the fixpoint `dExp`.

A similar idea of implementation follows for the commands:

```
Inductive Cmd : Type :=
 | skip : Cmd
 | sequence : Cmd → Cmd → Cmd
 | assign : Loc → Exp ℤ → Cmd
 | cond : Exp 𝔹 → Cmd → Cmd → Cmd
 | while : Exp bool → Cmd → Cmd
 | throw : EName → Cmd
 | try_catch : EName → Cmd → Cmd → Cmd.
```

```
Fixpoint dCmd (c: Cmd): (term unit unit) :=
 match c with
  | skip ⇒ (@id unit)
  | sequence c0 c1 ⇒ (dCmd c1) o (dCmd c0)
  | assign i a ⇒ (update i) o (dExp a)
  | cond b c2 c3 ⇒ copair (dCmd c2) (dCmd c3)
                    o (pbl o (dExp b))
  | while b c4 ⇒ (copair (lpd (pbl o (dExp b))
                         (dCmd c4) o (dCmd c4))
                         (@id unit)) o (pbl o (dExp b))
  | throw e ⇒ (throw unit e)
  | try_catch e c1 c2 ⇒ (@TRY_CATCH (dCmd c1)
                            (dCmd c2))
 end.
```

Fig. 27. IMP+Exc commands in Coq

In Fig. 27 on the left, we inductively define commands and on the right, recursively translate their behaviors into decorated settings. This translation is similar to the one given in Fig. 23, but this time done in Coq terms. Within

the above context, we retain sufficient material to prove equivalences among programs involving not only the state but also the exception effect.

6.1 Program equivalence proofs: the state and the exception

Here, we exemplify a bunch of program equivalence proofs. Note that for the sake of simplicity, we will use $\mathtt{u_x}$, $\mathtt{l_x}$, $(\mathtt{t\ op})$ and $(\mathtt{c\ p})$ instead of $(\mathtt{update\ x})^{(2,0)}$, $(\mathtt{lookup\ x})^{(1,0)}$, $(\mathtt{tpure\ op})^{(0,0)}$ and $(\mathtt{constant\ p})^{(0,0)}$, respectively.

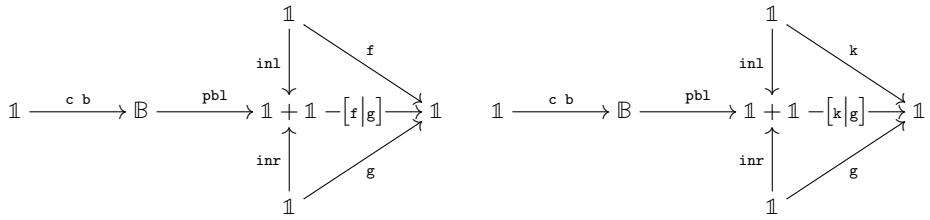
Remark 1. IMP specific properties of the state are slightly different than their generic versions given in Fig. 12. The ones we use through the following proofs are re-stated below. The full certified proofs can be found in the Coq release: see the given link at the end of the section.

1. $\mathtt{interaction\ update\text{-}update}\ \forall\, \mathtt{x} \in \mathtt{Loc\ p, q} : \mathbb{Z},\ \mathtt{u_x} \circ (\mathtt{c\ p}) \circ \mathtt{u_x} \circ (\mathtt{c\ q}) \equiv \mathtt{u_x} \circ (\mathtt{c\ p})$
2. $\mathtt{commutation\ update\text{-}update}\ \forall\, \mathtt{x} \neq \mathtt{y} \in \mathtt{Loc\ p, q} : \mathbb{Z},\ \mathtt{u_x} \circ (\mathtt{c\ p}) \circ \mathtt{u_y} \circ (\mathtt{c\ q}) \equiv \mathtt{u_y} \circ (\mathtt{c\ q}) \circ \mathtt{u_x} \circ (\mathtt{c\ p})$
3. $\mathtt{commutation\text{-}lookup\text{-}constant\text{-}update}\ \forall\, \mathtt{x} \in \mathtt{Loc, p, q} \in \mathbb{Z},\ \langle \mathtt{l_x}, (\mathtt{c\ q}) \rangle \circ \mathtt{u_x} \circ (\mathtt{c\ p}) \equiv \langle (\mathtt{c\ p}), (\mathtt{c\ q}) \rangle \circ \mathtt{u_x} \circ (\mathtt{c\ p})$

Fig. 28. Primitive properties of the state: IMP specific

Lemma 1. *For each* $\mathtt{f}^{(2,0)}, \mathtt{g}^{(2,0)} : \mathtt{Cmd}$ *and* $\mathtt{b}^{(0,0)} : \mathtt{bool}$*, let* $\mathtt{prog1} = (\mathtt{if\ b\ then\ f\ else\ g})$ *and* $\mathtt{prog2} = (\mathtt{if\ b\ then\ (if\ b\ then\ f\ else\ g)\ else\ g})$*. Then* $\mathtt{prog1} \equiv\!\equiv \mathtt{prog2}$*.*

Proof. We first sketch the diagrams of both programs as below:



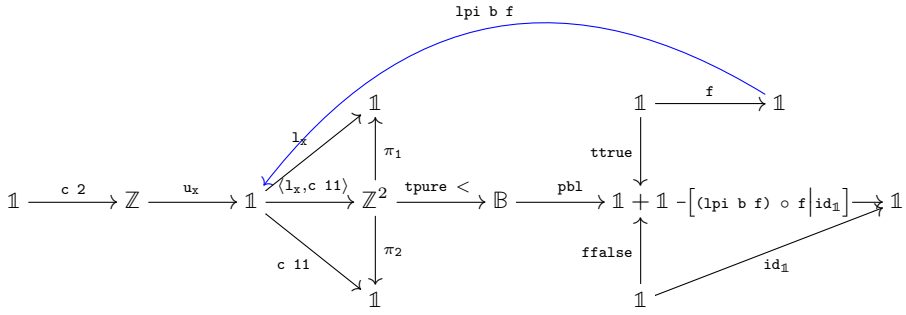
where $\mathtt{k} = (\mathtt{if\ b\ then\ f\ else\ g})$. Thus, $\big[\mathtt{f}\big|\mathtt{g}\big] \circ \mathtt{pbl} \circ \mathtt{c\ b} \equiv \big[\mathtt{k}\big|\mathtt{g}\big] \circ \mathtt{pbl} \circ \mathtt{c\ b}$. The proof proceeds by the induction on $\mathtt{b}$. If $\mathtt{b = false}$, by unfolding $\mathtt{pbl}$ and $(\mathtt{c\ false})$, we have $\big[\ \mathtt{f}\ \big|\ \mathtt{g}\ \big] \circ \mathtt{tpure}(\mathtt{bool\_to\_two}) \circ \mathtt{tpure}(\lambda \mathtt{x} : \mathtt{unit.false}) \equiv \big[\ \mathtt{k}\ \big|\ \mathtt{g}\ \big] \circ \mathtt{tpure}(\mathtt{bool\_to\_two}) \circ \mathtt{tpure}(\lambda \mathtt{x} : \mathtt{unit.false})$. We rewrite (imp_6) to get $\big[\ \mathtt{f}\ \big|\ \mathtt{g}\ \big] \circ \mathtt{tpure}(\lambda \mathtt{x} : \mathtt{unit.bool\_to\_two\ false}) \equiv \big[\ \mathtt{k}\ \big|\ \mathtt{g}\ \big] \circ \mathtt{tpure}\ (\lambda \mathtt{x} : \mathtt{unit.bool\_two\ false})$. Now, we cut $\mathtt{tpure}\ (\lambda \mathtt{x} : \mathtt{unit.bool\_to\_two\ false}) \equiv\!\equiv \mathtt{inr}$. So that we obtain $\big[\ \mathtt{f}\ \big|\ \mathtt{g}\ \big] \circ \mathtt{inr} \equiv\!\equiv \big[\ \mathtt{k}\ \big|\ \mathtt{g}\ \big] \circ \mathtt{inr}$. Then, we use (copair_2), and finally have $\mathtt{g} \equiv\!\equiv \mathtt{g}$. It remains to show $\mathtt{tpure}\ (\lambda \mathtt{x} : \mathtt{unit}.\ \mathtt{bool\_to\_two\ false}) \equiv\!\equiv \mathtt{inr}$. By simplifying $\mathtt{tpure}\ (\lambda \mathtt{x} : \mathtt{unit}.\ \mathtt{bool\ \_to\ \_two\ false})$ and unfolding $\mathtt{inr}$, we have $\mathtt{tpure}\ (\lambda \mathtt{x} : \mathtt{unit}.\ \mathtt{inr\ x}) \equiv\!\equiv (\mathtt{tpure\ inr})$. Now, we apply (imp_7) and get $\forall \mathtt{x} : \mathtt{unit}, \mathtt{inr\ x} = \mathtt{inr\ x}$.

Else if $\mathtt{b = true}$, by following above procedure with $\mathtt{true}$ (instead of $\mathtt{false}$) we first handle $\big[\mathtt{f}\big|\mathtt{g}\big] \circ \mathtt{inl} \equiv\!\equiv \big[\mathtt{k}\big|\mathtt{g}\big] \circ \mathtt{inl}$ and then freely convert $\equiv\!\equiv$ into $\equiv\!\sim$.

There, rewriting the rule (copair$_1$) yields $\mathtt{f} \equiv\sim \mathtt{k}$. We unfold $\mathtt{k}$ with $\mathtt{b} = \mathtt{true}$. Thus $\mathtt{f} \equiv\sim \left[\ \mathtt{f}\ \middle|\ \mathtt{g}\ \right] \circ \mathtt{inl}$. Now by rewriting (copair$_1$), we have $\mathtt{f} \equiv\sim \mathtt{f}$. □

Lemma 2. *For each* $\mathtt{x} : \mathtt{loc}$, *let* $\mathtt{prog3} = (\mathtt{x} := 2;\ \mathtt{while}\ (\mathtt{x} < 11)\ \mathtt{do}\ \mathtt{x} := \mathtt{x} + 4;)$ *and* $\mathtt{prog4} = (\mathtt{x} := 14)$. *Then* $\mathtt{prog3} \equiv\equiv\ \mathtt{prog4}$.

Proof. In the proof structure, we first deal with the pre-loop assignments and the looping pre-condition. Since it evaluates into *true*, in the second step we identify things related to the first loop iteration. The third step primarily studies the second and then the third loop iteration after which the looping pre-condition switches to *false*. Finally, we explain the program termination. Let us sketch the diagram of prog3:



where $\mathtt{f} = (\mathtt{x} := \mathtt{x} + 4)$ and $\mathtt{b} = (\mathtt{x} < 11)$.

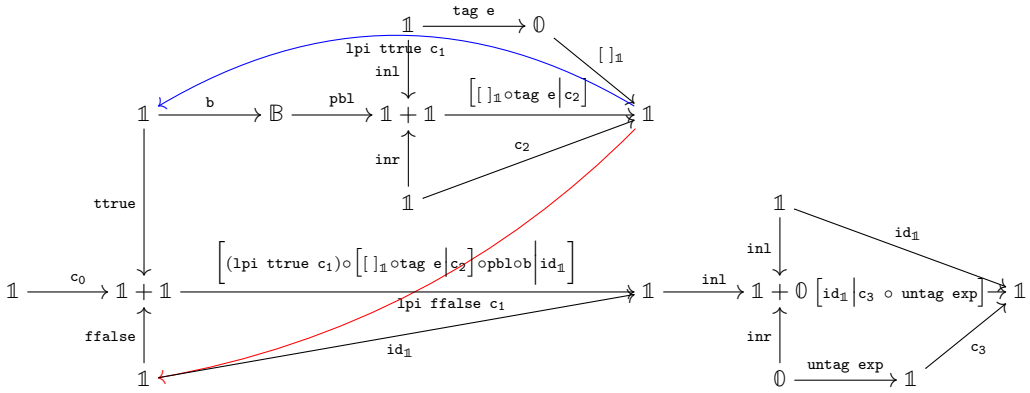
1. So that we have $\left[\ (\mathtt{lpi\ b\ f}) \circ \mathtt{f}\ \middle|\ \mathtt{id}_{\mathbb{1}}\ \right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <) \circ \langle \mathtt{l_x}, (\mathtt{c}\ 11)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. Let us try to simplify it as far as possible. By $\mathtt{commutation} - \mathtt{lookup} - \mathtt{constant} - \mathtt{update}$, we obtain $\left[\ (\mathtt{lpibf}) \circ \mathtt{f}\ \middle|\ \mathtt{id}_{\mathbb{1}}\ \right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <) \circ \langle (\mathtt{c}\ 2), (\mathtt{c}\ 11)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. By rewriting (imp$_2$): $\left[(\mathtt{lpi\ b\ f}) \circ \mathtt{f} \middle| \mathtt{id}_{\mathbb{1}}\right] \circ \mathtt{ttrue} \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. We first convert $\equiv\equiv$ into $\equiv\sim$ and then rewrite (copair$_1$). So that we have $(\mathtt{lpi\ b\ f}) \circ \mathtt{f} \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 14)$ which unfolds $(\mathtt{lpi\ b\ f}) \circ \mathtt{u_x} \circ (\mathtt{tpure}\ +) \circ \langle \mathtt{l_x}, \mathtt{c}\ 4\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 14)$. Since, there is no exceptional case, we are back to $\equiv\equiv$. By rewriting $\mathtt{commutation} - \mathtt{lookup} - \mathtt{constant} - \mathtt{update}$, we obtain $(\mathtt{lpi\ b\ f}) \circ \mathtt{u_x} \circ (\mathtt{tpure}\ +) \circ \langle \mathtt{c}\ 2, \mathtt{c}\ 4\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. The rule (imp$_2$) gives $(\mathtt{lpi\ b\ f}) \circ \mathtt{u_x} \circ (\mathtt{c}\ 6) \circ \mathtt{u_x} \circ (\mathtt{c}\ 2) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. Now, by the lemma $\mathtt{interaction\text{-}update\text{-}update}$, we get $(\mathtt{lpi\ b\ f}) \circ \mathtt{u_x} \circ (\mathtt{c}\ 6) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$.
2. We can rewrite (imp-loopiter) and get $\left[\ (\mathtt{lpi\ b\ f}) \circ \mathtt{f}\ \middle|\ \mathtt{id}_{\mathbb{1}}\ \right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <) \circ \langle \mathtt{l_x}, (\mathtt{c}\ 11)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 6) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. In the second iteration with the above procedure, we have $\left[\ (\mathtt{lpi\ b\ f}) \circ \mathtt{f}\ \middle|\ \mathtt{id}_{\mathbb{1}}\ \right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <) \circ \langle \mathtt{l_x}, (\mathtt{c}\ 11)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 10) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$.
3. The third iteration yields $\left[\ (\mathtt{lpi\ b\ f}) \circ \mathtt{f}\ \middle|\ \mathtt{id}_{\mathbb{1}}\right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <) \circ \langle \mathtt{l_x}, (\mathtt{c}\ 11)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 14) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c}\ 14)$. Now; again by rewriting the lemma $\mathtt{commutation\text{-}lookup\text{-}constant\text{-}update}$, we have $\left[\ (\mathtt{lpi\ b\ f}) \circ \mathtt{f}\ \middle|\ \mathtt{id}_{\mathbb{1}}\right] \circ$

$\texttt{pbl} \circ (\texttt{tpure} <) \circ \langle (\texttt{c}\ 14), (\texttt{c}\ 11) \rangle \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 14) \equiv\equiv \texttt{u}_\texttt{x} \circ (\texttt{c}\ 14)$. We rewrite ($\text{imp}_2$) and then obtain $\left[\ (\texttt{lpi b f}) \circ \texttt{f}\ \middle|\ \texttt{id}_\mathbb{1}\right] \circ \texttt{inr} \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 14) \equiv\equiv \texttt{u}_\texttt{x} \circ (\texttt{c}\ 14)$.

4. Finally, it suffices to rewrite (copair_2); $\texttt{id}_\mathbb{1} \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 14) \equiv\equiv \texttt{u}_\texttt{x} \circ (\texttt{c}\ 14)$. □

Lemma 3. *For each* $\texttt{x}\ \texttt{y} : \texttt{Loc}$, $\texttt{e} : \texttt{EName}$, *let* $\texttt{prog5} = (\texttt{x} := 1;\ \texttt{y} := 20;\ \texttt{try}\ ((\texttt{while}\ (\texttt{tt})\ \texttt{do}\ (\texttt{if}\ (\texttt{x} <= 0)\ \texttt{then}\ (\texttt{throw e})\ \texttt{else}\ (\texttt{x} := \texttt{x} - 1))))\ \texttt{catch}\ (\texttt{e} \Rightarrow (\texttt{y} := 7)))$ *and* $\texttt{prog6} = (\texttt{x} := 0; \texttt{y} := 7)$. *Then* $\texttt{prog5} \equiv\equiv\ \texttt{prog6}$.

Proof. Within the below enumerated proof structure, we first tackle with the `downcast` operator. The second task is to deal with the first loop iteration which has the state but no exception effect. In the third, we study the second iteration of the loop where an exception is thrown. Finally, in the fourth step, we explain the loop termination followed by the exception recovery and the program termination. Let us now sketch the diagram of `prog5`:



where $\texttt{b} = (\texttt{x} <= 0)$, $\texttt{c}_0 = (\texttt{x} := 0; \texttt{y} := 20)$, $\texttt{c}_1 = (\texttt{if}(\texttt{x} <= 0)\ \texttt{then}(\texttt{throw e})\ \texttt{else}\ (\texttt{x} := \texttt{x} - 1))$, $\texttt{c}_2 = (\texttt{x} := \texttt{x} - 1)$ and $\texttt{c}_3 = (\texttt{y} := 7)$.

1. We have $\downarrow \left(\left[\ \texttt{id}_\mathbb{1}\ \middle|\ \texttt{c}_3 \circ \texttt{untag e}\ \right] \circ \texttt{inl} \circ \left[\ (\texttt{lpi ttrue c}_1) \circ \left[\ [\,]_\mathbb{1} \circ \texttt{tag e}\ \middle|\ \texttt{c}_2\ \right] \circ \texttt{pbl} \circ \texttt{b}\ \middle|\ \texttt{id}_\mathbb{1}\ \right] \circ \texttt{ttrue} \right) \circ \texttt{u}_\texttt{y} \circ (\texttt{c}\ 20) \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 1) \equiv\equiv \texttt{u}_\texttt{y} \circ (\texttt{c}\ 7) \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 0)$. We first convert $\equiv\equiv$ into $\equiv\sim$, then rewrite the ($\text{downcast}_\sim$) rule and get $\left[\ \texttt{id}_\mathbb{1}\ \middle|\ \texttt{c}_3 \circ \texttt{untag e}\ \right] \circ \texttt{inl} \circ \left[\ (\texttt{lpi ttrue c}_1) \circ \left[\ [\,]_\mathbb{1} \circ \texttt{tag e}\ \middle|\ \texttt{c}_2\ \right] \circ \texttt{pbl} \circ \texttt{b}\ \middle|\ \texttt{id}_\mathbb{1}\ \right] \circ \texttt{ttrue} \circ \texttt{u}_\texttt{y} \circ (\texttt{c}\ 20) \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 1) \equiv\sim \texttt{u}_\texttt{y} \circ (\texttt{c}\ 7) \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 0)$. Rewriting `commutation-update-update`, on both sides, gives $\left[\texttt{id}_\mathbb{1} \middle| \texttt{c}_3 \circ \texttt{untag e}\right] \circ \texttt{inl} \circ \left[(\texttt{lpi ttrue c}_1) \circ \left[\ [\,]_\mathbb{1} \circ \texttt{tag e}\ \middle|\ \texttt{c}_2\ \right] \circ \texttt{pbl} \circ \texttt{b}\ \middle|\ \texttt{id}_\mathbb{1}\ \right] \circ \texttt{ttrue} \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 1) \circ \texttt{u}_\texttt{y} \circ (\texttt{c}\ 20) \equiv\sim \texttt{u}_\texttt{x} \circ (\texttt{c}\ 0) \circ \texttt{u}_\texttt{y} \circ (\texttt{c}\ 7)$.
2. Now; we rewrite the rule (copair_1), and handle $\left[\ \texttt{id}_\mathbb{1}\ \middle|\ \texttt{c}_3 \circ \texttt{untag e}\ \right] \circ \texttt{inl} \circ (\texttt{lpi ttrue c}_1) \circ \left[\ [\,]_\mathbb{1} \circ \texttt{tag e}\ \middle|\ \texttt{c}_2\ \right] \circ \texttt{pbl} \circ \texttt{b} \circ \texttt{u}_\texttt{x} \circ (\texttt{c}\ 1) \circ \texttt{u}_\texttt{y} \circ$

$(\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0)$. By unfolding b, we get $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \left[\ [\,]_{\mathbb{1}} \circ \mathtt{tag\ e}\ \middle|\ \mathtt{c_2}\ \right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <=) \circ \langle \mathtt{l_x}\ (\mathtt{c}\ 0)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. With the help of lemma commutation− lookup− constant− update, we obtain $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \left[\ [\,]_{\mathbb{1}} \circ \mathtt{tag\ e}\ \middle|\ \mathtt{c_2}\ \right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <=) \circ \langle (\mathtt{c}\ 1), (\mathtt{c}\ 0)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. The rule (imp_2) gives $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \left[\ [\,]_{\mathbb{1}} \circ \mathtt{tag\ e}\ \middle|\ \mathtt{c_2}\ \right] \circ \mathtt{ffalse} \circ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y}. \circ (\mathtt{c}\ 7)$. We now rewrite (copair_2) $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{c_2} \circ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. Here, we unfold $\mathtt{c_2}$, $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{u_x} \circ (\mathtt{tpure}\ -) \circ \langle \mathtt{l_x}, (\mathtt{c}\ 1)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. The lemma $\mathtt{commutation} - \mathtt{lookup} - \mathtt{constant} - \mathtt{update}$ gives $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{u_x} \circ (\mathtt{tpure}\ -) \circ \langle (\mathtt{c}\ 1), (\mathtt{c}\ 1)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ 0(\mathtt{c}\ 7)$. We rewrite (imp_1) and then get $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{u_x} \circ (\mathtt{c}\ 0)\ \mathtt{u_x} \circ (\mathtt{c}\ 1) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. We again rewrite the lemma commutation-update-update and obtain $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x}\circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$.

3. We re-iterate the loop via (imp-loopiter) with $\mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$: $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ \left[(\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{c_1}\ \middle|\ \mathtt{id}\ \right] \circ \mathtt{ttrue} \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. We first rewrite (copair_1) and unfold $\mathtt{c_1}$: $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \left[\ \mathtt{throw\ e}\ \mathbb{1}\ \middle|\mathtt{c_2}\right] \circ \mathtt{pbl} \circ (\mathtt{tpure}\ <=) \circ \langle \mathtt{l_x}, (\mathtt{c}\ 0)\rangle \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$. By rewriting $\mathtt{commutation} - \mathtt{lookup} - \mathtt{constant} - \mathtt{update}$ and (imp_3), the comparison yields in ttrue. So that: $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\mathtt{lpi\ ttrue\ c_1}) \circ \left[\ \mathtt{throw\ e}\ \mathbb{1}\ \middle|\ \mathtt{c_2}\ \right] \circ \mathtt{ttrue} \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$. By (copair_1), the exception is thrown: $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ \big((\mathtt{lpi\ ttrue\ c_1}) \circ \mathtt{throw\ e}\ \mathbb{1}\ \big) \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$. Now; via interaction-propagator-throw, we get $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ (\ \mathtt{throw\ e}\ \mathbb{1}\ ) \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$.

4. Here, we first unfold throw: $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inl} \circ [\,]_{\mathbb{1}} \circ \mathtt{tag\ e} \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$ then, cut $\mathtt{inl} \circ [\,]_{\mathbb{1}} \equiv\equiv \mathtt{inr}$. Thus, we have $\left[\ \mathtt{id}_{\mathbb{1}}\ \middle|\ \mathtt{c_3} \circ \mathtt{untag\ e}\ \right] \circ \mathtt{inr} \circ \mathtt{tag\ e} \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. By (copair_2), $\mathtt{c_3} \circ \mathtt{untag\ e} \circ \mathtt{tag\ e} \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. Since $\mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20)$ is pure with respect to the exception, we rewrite (eax_1) to get $\mathtt{c_3} \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. If follows $\mathtt{c_3} = (\mathtt{u_y} \circ (\mathtt{c}\ 7))$ that $\mathtt{u_y} \circ (\mathtt{c}\ 7) \circ \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. We now rewrite commutation− update− update on the left to have $\mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7) \circ \mathtt{u_y} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. Finally, it suffices to rewrite interaction− update−update, $\mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7) \equiv\sim \mathtt{u_x} \circ (\mathtt{c}\ 0) \circ \mathtt{u_y} \circ (\mathtt{c}\ 7)$. It still remains to prove that $\mathtt{inl} \circ [\,]_{\mathbb{1}} \equiv\equiv \mathtt{inr}$: since

everything is pure with respect to the exception, we have $\texttt{inl} \circ [\,]_{\mathbb{1}} \equiv_{\sim} \texttt{inr}$. Now, rewriting the rule $(\text{unit}_{\sim})$ suffices to have $[\,]_{\mathbb{1}+\mathbb{1}} \equiv_{\sim} [\,]_{\mathbb{1}+\mathbb{1}}$. □

The complete Coq library with all certified proofs can be found on https://forge.imag.fr/frs/download.php/651/IMP-STATES-EXCEPTIONS-0.3.tar.gz.

7 Conclusion

We have presented new frameworks for formalizing the treatment of the state and the exception via the decorated logic both separately and combined with Coq implementations. Decorations form a bridge between the syntax and the interpretation by turning the syntax sound without adding any explicit *type of the state* nor *the exception*. Combined setting is specialized for the IMP+Exc language and finally equivalence proofs of programs are given with related certifications in Coq. Besides, in [5], we prove that the core language for the state and exception as well as the programmers' language for the exception are complete.

References

1. Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. Decorated proofs for computational effects: States. ACCAT 2012. Electronic Proceedings in Theoretical Computer Science 93, p. 45-59 (2012).
2. Jean-Guillaume Dumas, Dominique Duval, Laurent Fousse, Jean-Claude Reynaud. A duality between exceptions and states. Mathematical Structures in Computer Science 22, p. 719-722 (2012).
3. Jean-Guillaume Dumas, Dominique Duval, Burak Ekici and Jean-Claude Reynaud. Certified proofs in programs involving exceptions. CICM 2014, CEUR Workshop Proceedings, n^o 1186, paper 20.
4. Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous. Formal verification in Coq of program properties involving the global state. JFLA 2014, pages 1–15, January 2014.
5. Jean-Guillaume Dumas, Dominique Duval, Burak Ekici, Damien Pous and Jean-Claude Reynaud. Hilbert-Post completeness for the state and the exception effects. Research report (2015).
6. César Dominguez, Dominique Duval. Diagrammatic logic applied to a parameterization process. Mathematical Structures in Computer Science 20(04) p. 639-654.
7. Claude Marché. MPRI Course notes: Proof of a program. (2012).
8. John M. Lucassen, David K. Gifford. Polymorphic effect systems. POPL 1988. ACM Press, p. 47-57.
9. Eugenio Moggi. Notions of Computation and Monads. Information and Computation 93(1), p. 55-92 (1991).
10. Gordon D. Plotkin, John Power. Notions of Computation Determine Monads. FoSSaCS 2002. LNCS, Vol. 2620, p. 342-356, Springer (2002).
11. Gordon D. Plotkin, Matija Pretnar. Handlers of Algebraic Effects. ESOP 2009. LNCS, Vol. 5502, p. 80-94, Mpringer (2009).
12. Sam Staton. Completeness for Algebraic Theories of Local State. FoSSaCS 2010. LNCS, Vol. 6014, p. 48-63, Springer (2010).
13. Viviana Bono, Manfred Kerber. Extending Hoare Calculus to Deal with Crash. The University of Birmingham, School of Computer Science, CSR-06-08.