IMP with exceptions over decorated logic
Burak Ekici

To cite this version:
Burak Ekici. IMP with exceptions over decorated logic. Discrete Mathematics and Theoretical Computer Science, DMTCS, 2018, vol. 20 no. 2. hal-01132831v9

HAL Id: hal-01132831
https://hal.archives-ouvertes.fr/hal-01132831v9
Submitted on 12 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
IMP with exceptions over decorated logic

Burak Ekici

University of Innsbruck, Department of Computer Science, Innsbruck, Austria


In this paper, we facilitate the reasoning about impure programming languages, by annotating terms with “decorations” that describe what computational (side) effect evaluation of a term may involve. In a point-free categorical language, called the “decorated logic”, we formalize the mutable state and the exception effects first separately, exploiting a nice duality between them, and then combined. The combined decorated logic serves as the target language for the denotational semantics of the IMP+Exc imperative programming language, and allows us to prove equivalences between programs written in IMP+Exc. The combined logic is encoded in Coq, and this encoding is used to certify some program equivalence proofs.

Keywords: Computational effects, state, exceptions, program equivalence proofs, decorated logic, Coq.

1 Introduction

In programming languages theory, a program is said to have computational effects if, besides a return value, it has observable interactions with the outside world. For instance, using/modifying the program state, raising/recovering exceptions, reading/writing data from/to some file, etc. In order to formally reason about behaviors of a program with computational effects, one has to take into account these interactions. One difficulty in such a reasoning is the mismatch between the syntax of operations with effects and their interpretation. Typically, an operation in an effectful language with arguments in $X$ that returns a value in $Y$ is not interpreted as a function from $X$ to $Y$, due to the effects, unless the operation is pure.

The best known algebraic approach to formalize computational effects was initiated by Moggi (1991) in his seminal paper. He showed that the effectful operations of an impure language can be interpreted as arrows of a Kleisli category for an appropriate monad $(T, \eta, \mu)$ over a base category $\mathcal{E}$ with finite products. For instance, in Moggi’s computational metalanguage, an operation in an impure language with arguments in $X$ that returns a value in $Y$ is now interpreted as an arrow from $\|X\|$ to $T\|Y\|$ in $\mathcal{E}$ where $\|X\|$ is the object of values of type $X$ and $T\|Y\|$ is the object of computations that return values of type $Y$. The use of monads to formalize effects (such as state, exceptions, input/output and non-deterministic choice) was popularized by Wadler (1992), and implemented in the programming languages Haskell and F#. Using monad transformers, as in Jaskelioff (2009), it is usually possible to “combine” different effects formalized by monads. Moggi’s computational metalanguage was extended into the basic effect calculus with a notion of computation type by Filinski (1996) in his effect PCF and by Levy (1999) in his call-by-push-value (CBPV). Egger et al. (2014) defined their effect calculus, named extended effect calculus as a canonical calculus incorporating the ideas of Moggi, Filinski and Levy. Following Moggi, they included a type...
constructor for computations. Following Filinski and Levy, they classified types into value types and computation types.

Being dual to monads, comonads have been used to formalize context-dependent computations. Intuitively, an effect which observes features may arise from a comonad, while an effect which constructs features may arise from a monad (Jacobs and Rutten (2011)). Uustalu and Vene (2008) have structured stream computations, Orchard et al. (2010) array computations and Tzevelekos (2008) game semantics via the use of comonads. Petriček et al. proposed a unified calculus for tracking context dependence in functional languages together with a categorical semantics based on indexed comonads. In (Orchard (2012)), there is a quite nice discussion on how to choose between a monad or comonad when either can be used to capture a particular notion of computation. Also, Brookes and Van Stone (1993) discussed that a computation may be interpreted by distributive laws of a comonad over a monad when it is seen as a composition of context-dependence and effectfulness. This approach has been applied to clocked causal data-flow computation, combining causal data-flow and exceptions by Uustalu and Vene (2005).

Moggi’s approach, using monads in effect modeling, has been extended to Lawvere theories which first appeared in Lawvere (1963)’s PhD dissertation. Then, Linton (1966, 1969) first showed that every Lawvere theory induces a monad on the category of sets, and then on any category satisfying some condition called the “local representability”. Therefore, Moggi’s seminal idea, formalizing computational effects by monads, made it possible for monadic effects to be formalized through Lawvere theories. To this extend, Plotkin and Power (2002) have shown that effects such as the global and the local state could be formalized by signatures of effectful terms and an equational theory explaining the interactions between them. Melliès (2010) has refined this equational theory showing that some of the equations modeling the mutable global state can be omitted. Hyland et al. (2006, 2007) studied the combination of computational effects in terms of Lawvere theories.

Plotkin and Pretnar (2009, 2013) extended Moggi’s classification of terms (values and computations) with a third level called handlers for the computational effects that can be represented by an algebraic theory (algebraic effects). Initially, they introduce an handler for the exception handling, and then account for its generalization to the other handlers to cope with other algebraic effects such as stream redirection, explicit non-determinism, CCS, parameter passing, timeout and rollback (Plotkin and Pretnar, 2013, §3). For each algebraic effect, handling constructs are used to apply handlers to effectful computations where effectful computations can be interpreted as algebraic operations while handling constructs as homomorphisms from free algebras. This use of handling constructs is inspired from Benton and Kennedy (2001)’s work where a single construct specialized to handle exceptions is introduced. Moreover, Jacobs (2001) formalized the exception effect from the dual, namely co-algebraic, viewpoint. Exception handling is also used to build a Hoare logic for exceptions by Schröder and Mossakowski (2004).

There is an older formal way of modeling computational effects called the effect systems by Lucassen and Gifford (1988). They presented an approach to programming languages for parallel computers. The key idea was to use an effect system to discover expression scheduling constraints. There, every expression comes with three components: types to represent the kinds of the return values, effects to summarize the observable interactions of expressions and regions to highlight the areas of the memory where expressions may have effects. To this extend, one can simply reason that if two expressions do not have overlapping effects, then they can obviously be scheduled in parallel. The reasoning is done by some inference rules for types and effects based on the second order typed \( \lambda \) -calculus.

Domínguez and Duval (2010) proposed yet another paradigm to formalize computational effects by mixing effect systems and algebraic theories, named the decorated logic. The key point of this paradigm is
that every term comes with a decoration which exposes its features with respect to a single computational
effect or to several ones keeping their interpretations close to syntax in reasoning with effects. In addition,
an equational theory highlights the interactions among terms with two sorts of equations: weak equations
relate terms with respect only to their results while strong equations relate them with respect both to their
results and effects. By and large, decorated logic provides an equational reasoning in between programs
written in imperative languages after being used as a target language for a denotational semantics of the
studied language.

In a decorated logic, a term has three different decorations: pure, accessor and modifier/catcher. The
first two decorations can correspond to Moggi’s values and computations, and the third level can be seen
as Plotkin and Pretnar’s handlers. An handler operates recursively by its nature, and handles also the
continuation. However, a catcher does not. It returns the continuation unhandled which should then be
handled explicitly. Thus, catchers are non-recursive handlers, so called shallow handlers introduced by
Kammar et al. (2013).

1.1 On the use of decorated logic

In this paper, we use Duval’s decorated logic to formalize computational effects. The advantages of
using decorated logic in effect formalization is mainly two-folded: (1) effects of terms are hidden by
the decorations, so that it is possible to preserve the syntax of term signatures. Thereafter, the provided
equational reasoning would be valid for different algebraic models of the same effect. (2) The equational
theory is based on decorated equivalence relations proposing different reasoning capabilities: one on
effects and returned results and the other one only on returned results. However, for the time being, it
might be inconvenient to use decorated logic to prove more general properties of algorithms. That is to
say, we can prove equivalences between programs that admits particular specifications as initializing and
describing final values stored in variables. The total correctness of a theory in a decorated logic, that
guarantees that the theory is not using too many axioms to become the maximal theory, is based on a
syntactic completeness property called relative Hilbert-Post completeness. Section 7.3 details mentioned
property, and its application to the specific case that this paper covers.

1.2 Organization and contributions

In general terms, in this paper, we extend Moggi’s original approach using the classifications of expressions,
provided by the Kleisli category of the monad of exception and the comonad of the state thanks to the
duality between states and exceptions proven by Dumas et al. (2012). The definitions and the results are
presented in terms of equational theories so that one does not need to know the details about the monad of
exceptions nor the comonad of the state. In more specific terms, this paper designs the decorated logic
for the global state and the exception effects, and then combines them to serve as a target language for
denotational semantics of imperative programming languages mixing mentioned effects. It is organized
as follows: in Section 2, we introduce an imperative programming language that mixes the state and the
exception effects by defining its small-step operational semantics. The language we study there is called
IMP+Exc which extends the IMP (or while) with a mechanism to raise and handle exceptions. In Section 3,
we introduce the decorated logic as a generic framework extending Moggi’s monadic equational logic.
Then, we formally specialize the decorated logic for the state and the exception effects in Sections 4 and 5,
respectively. In Section 6, we combine these logics. Finally, Section 7 details the use of the combined
decorated logic as the target language for the IMP+Exc denotational semantics. This provides a rigorous
formalism for an equational reasoning between termination-guaranteed IMP+Exc programs. I.e., proving
two different looking programs are in fact doing the same job with respect to the state and exception effects. In Section 7.1, we presents three proof examples. Also, we certify such proofs with the Coq Proof Assistant. See the entire Coq implementation here \(^{(i)}\), and the approach of the paper in Figure 1.

This paper builds upon several papers by Domínguez and Duval (2010), Dumas et al. (2014a), Dumas et al. (2015), Dumas et al. (2014b), Dumas et al. (2012) and Dumas et al. (2014c). The novel points presented here can be itemized as follows:

- a combined decorated logic for the states (Dumas et al. (2014a)) and exceptions (Dumas et al. (2014b)) effects (this paper explains both logics again but for the details please refer to the citations),
- Coq formalization of the combined logic,
- a denotational semantics for the IMP+Exc (IMP with exceptions) over the combined logic,
- Coq formalization of the IMP+Exc denotational semantics,
- some equivalence proofs of programs written in IMP+Exc and their verifications in Coq.

A preliminary version of this paper has been presented in TFP (Trends in Functional Programming) 2015 but did not appear in the final proceedings. Find the mentioned paper here\(^{(ii)}\).

2 IMP with exceptions

**IMP** is a standard Turing complete imperative programming language natively providing global variables of integer (\(\mathbb{Z}\)), Boolean (\(\mathbb{B}\)) and unit (\(\mathbb{U}\)) data types, standard integer and Boolean arithmetic enriched with a set of commands that is made of do-nothing, assignment, sequence, conditionals and looping operations. Below, we detail its syntax where \(n\) represents a constant integer term while \(x\) is an integer global variable. Note also that abbreviations \(aexp\) and \(bexp\) respectively denote arithmetic and Boolean expressions as well as \(cmd\) stands for the commands.

\[
\begin{align*}
aexp &: n | x | a_1 + a_2 | a_1 - a_2 | a_1 \times a_2 \\
bexp &: true | false | a_1 = a_2 | a_1 \neq a_2 | a_1 > a_2 | a_1 < a_2 | b_1 \land b_2 | b_1 \lor b_2 | \neg b_1 \\
cmd &: SKIP | x \triangleq a_1 | c_1; c_2 | if b then c_1 else c_2 | while b do c_1
\end{align*}
\]

\(^{(i)}\) https://github.com/ekiciburak/impex-on-decorated-logic
Neither arithmetic nor Boolean expressions are allowed to modify the state: they are either pure or read-only. We present, in Figure 3, the big-step semantics for evaluation of arithmetic expressions in IMP where we use a big-step transition function \( \rightarrow_a : \text{aexp} \times \mathcal{S} \rightarrow \mathbb{Z} \). This function computes an integer value out of an input arithmetic expression and the current program state (denoted \( s \)) which includes contents of variables at a given time. The symbol \( \text{op} \) represents the operation symbols (+, − or ×) given by the

\[
\begin{align*}
(a\text{const}) & : (n, s) \rightarrow_{a} n \\
(\text{var}) & : (x, s) \rightarrow_{a} s(x) \\
(\text{op} - \text{sym}) & : (a_1, s) \rightarrow_{a} n_1 \quad (a_2, s) \rightarrow_{a} n_2 \\
& : (a_1 \text{ op } a_2, s) \rightarrow_{a} a_1 \text{ op } a_2 \text{ op } n_2
\end{align*}
\]

\textbf{Figure 3: Big-step operational semantics for arithmetic expressions}

standard syntax in Figure 3, while \( \text{op}_2 : \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \) denotes the corresponding binary operations in \( \mathbb{Z} \). Similarly, in Figure 4, we present the big-step semantics for evaluation of Boolean expressions in IMP where we use a big-step transition function \( \rightarrow_b : \text{bexp} \times \mathcal{S} \rightarrow \mathbb{B} \). This function simply computes a Boolean value out of an input Boolean expression and the current program state. The constant symbols \text{true} and \text{false} are Boolean operation symbols given by the standard syntax in Figure 2, while \text{true} and \text{false} are Boolean constructors. Similarly, \( \text{op} \) represents the binary operation symbols, while \( \text{op}_b : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B} \) denotes the corresponding Boolean operations, and \( \text{neg} : \mathbb{B} \rightarrow \mathbb{B} \) is the Boolean negation.

The small-step operational semantics for evaluation of commands are given in Figure 5 where we use a small-step transition function \( \rightsquigarrow : \mathcal{S} \times \mathcal{cmd} \rightarrow \mathcal{S} \times \mathcal{cmd} \) which is interpreted as \textit{at the state \( s \), one step execution of the command \( c \) changes the state into \( s' \) and the command \( c' \) is now in further execution}.

\[
\begin{align*}
(\text{sequence}) & : s, c_1 \rightsquigarrow s', c'_1 \\
&s, (c_1 ; c_2) \rightsquigarrow s', (c'_1 ; c'_2) \\
(\text{skip}) & : s, \text{(SKIP)} \rightsquigarrow s, \text{c} \\
(\text{assign}) & : (a, s) \rightarrow_{a} n \\
&s, (x := a) \rightsquigarrow s[x \leftarrow n], \text{SKIP} \\
(\text{cond}_1) & : (b, s) \rightarrow_{b} \text{true} \\
&s, (\text{if then } c_1 \text{ else } c_2) \rightsquigarrow s, (c_1) \\
(\text{cond}_2) & : (b, s) \rightarrow_{b} \text{false} \\
&s, (\text{if then } c_1 \text{ else } c_2) \rightsquigarrow s, (c_2) \\
(\text{while}_1) & : (b, s) \rightarrow_{b} \text{true} \\
&s, (\text{while } b \text{ do } c) \rightsquigarrow s, (c) \\
(\text{while}_2) & : (b, s) \rightarrow_{b} \text{false} \\
&s, (\text{while } b \text{ do } c) \rightsquigarrow s, \text{SKIP}
\end{align*}
\]

\textbf{Figure 5: Small-step operational semantics for commands}

We need to elucidate that a command \( c \) terminates at a state \( s' \) if \( s, c \rightsquigarrow s', \text{SKIP} \) for some state \( s' \),
where \( \rightarrow^* \) is the transitive closure of the transition function \( \rightarrow \). Mind also that \texttt{SKIP} is allowed to execute at any state \( s \), and \texttt{SKIP} alone is used to indicate the final step of some command set.

### 2.1 A mechanism to handle exceptions

Extending the \texttt{IMP} language with a mechanism that allows raising exceptions and recovering from them, we enrich the command set with \texttt{THROW} and \texttt{TRY/CATCH} blocks. In addition to the ones in Figure 2, we also consider following commands in Figure 6 where \( e \) is an exception name coming from a finite set \( \texttt{EName} \) which exists by assumption. There is also a type \( \texttt{EV}_e \) of exceptional values (parameters) for each exception name \( e \). The small-step operational semantics for \texttt{THROW} and \texttt{TRY/CATCH} commands are shown in Figure 7.

![Figure 6: Syntax for exceptional commands](image)

![Figure 7: Small-step operational semantics for additional commands](image)

Exceptional commands are pure in terms of 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 \texttt{SKIP} alone indicates the termination of a program. Now, we extend this by saying \texttt{THROW} \( e \) is also an end but an abnormal end. Intuitively, if an exceptional value of name \( e \) is raised in the \texttt{TRY} block and recovered immediately in the \texttt{CATCH}, the program then resumes with the provided continuation. An exceptional value (of name \( e \)) gets propagated if another exceptional value with different name (say, of name \( f \), s.t. \( e \neq f \)) is being recovered in the \texttt{CATCH}.

In Section 7, we define denotational semantics of the \texttt{IMP+Exc} language using the decorated logic (generic framework is given in Section 3) for the state and the exception effects as the target language. We present this logic in Section 6 as a combination of the logics that we introduce in Sections 4 and 5.

### 3 Decorated Logic (\( \mathcal{L}_{dec} \))

The decorated logic, as a generic framework, is an extension to monadic equational logic Moggi (1991), that we briefly discuss in Section 3.1, with the use of decorations on terms and equalities. It provides a rigorous formalism to do \emph{equational reasoning} between impure programs written in imperative programming languages with side effects after being defined as a target language for their denotational semantics.
3.1 Monadic Equational Logic ($\mathcal{L}_{meq}$)

The monadic equational logic ($\mathcal{L}_{meq}$) is the minimal logic that can be interpreted in a category with objects as types, arrows as terms and equalities as equations. I.e., an object $0$ in the category interprets the type $X$ in the logic, just as the usual Leibniz equality, $f = g$, interprets the equation $f \equiv g$ in the logic. The keyword “monadic” has little to do with monads. It rather means that the operations of the logic are unary (or mono-adic). Figure 8 presents the syntax of the logic $\mathcal{L}_{meq}$. There, every term has a source and a

target type, e.g., $f : X \to Y$. Every equation is formed by terms with the same source and target types, e.g., $e : f \equiv g$ where $f$, $g : X \to Y$. This syntax is accompanied by the rules shown in Figure 9.

**Grammar for the monadic equational logic:**

| Types: | $t ::= X | Y | \cdots$ |
| Terms: | $f, g ::= id_t \mid a \mid b \mid \cdots \mid g \circ f$ |
| Equations: | $eq ::= f \equiv g$ |

**Figure 8: $\mathcal{L}_{meq}$: syntax**

The congruence rules say that the relation '$\equiv$' is a congruence meaning that it is an equivalence relation (reflexive, symmetric and transitive) which obeys replacements and substitutions of compatible terms with respect to the composition. The basic categorical rules indicate that there is an identity morphism $id_X : X \to X$ for each type $X$, composition is an associative operation, and composing any term $f$ with $id$ is $f$, up to $\equiv$, no matter the composition order.

**Figure 9: $\mathcal{L}_{meq}$: rules**

3.2 The decorated logic

The decorated logic extends the monadic equational logic with a 3-tier effect system for terms and a 2-tier system for equations made of “up-to-effects” (weak) and “strong” equalities. Figure 10 presents its syntax. Each term has a source and a target type as well as a decoration which describe what computational side effects evaluation of that term may involve, and used as a superscript $(0)$, $(1)$ or $(2)$: a pure term is decorated with $(0)$, an effect constructor with $(1)$ and an effect modifier term comes with the decoration $(2)$. Each equation is formed by two terms with the same source and target as well as a decoration which is denoted either by “$\sim$” (weak) or by “$\equiv$” (strong). A weak equality between two terms relates them according only to their results, while a strong equality relates terms according both to their result and the side effect evaluations they involve with respect to the effect in question.
Grammar for the decorated logic:

Types: $t ::= X \mid Y \ldots$

Decoration for terms: $(d) ::= (0) \mid (1) \mid (2)$

Terms: $f, g ::= a^{(d)} \mid b^{(d)} \mid \ldots \mid g \circ f^{(d)} \mid (\text{tpure} \circ)^{(0)}$

Equations: $\text{eq} ::= f \equiv g \mid f \sim g$

Figure 10: $L_{\text{dec}}$: syntax

The $\text{tpure}$ is a special constructor used to introduce decorated pure terms into the logic $L_{\text{dec}}$. It inputs a non-decorated pure term from a pure type system (i.e., Coq’s logic) and drops it in with the decoration $(0)$. For instance, the identity term $\text{id}$ is defined using the $\text{tpure}$ constructor, for all types $X$ as follows:

$$\text{id}_{X}^{(0)} : X \rightarrow X ::= \text{tpure} (\lambda x : X. x : X).$$

In Figure 11, we present the inference rules associated to the syntax given in Figure 9.

Remark 3.1. In all of the figures presenting the rules of some decorated logic, throughout out the paper, the decorations “$d_1, d_2, d_3, \ldots$” are meant to be in the set $\{0, 1, 2\}$ unless otherwise stated. For instance, in the rule (wtos) below decorations $d_1$ and $d_2$ cannot take the value 2.

hierarchy rules

(0-to-1) $\frac{f^{(0)}}{f^{(1)}}$ (1-to-2) $\frac{f^{(1)}}{f^{(2)}}$ (stow) $\frac{f^{(d_1)} \equiv g^{(d_2)}}{f^{(d_1)} \sim g^{(d_2)}}$ (wtos) $\frac{f^{(d_1)} \sim g^{(d_2)}}{d_1, d_2 \in \{0, 1\}}$

congruence rules

(refl) $\frac{f^{(d_1)}}{f^{(d_1)} \equiv f^{(d_1)}}$ (sym) $\frac{g^{(d_2)} \equiv f^{(d_1)}}{f^{(d_1)} \sim g^{(d_2)}}$ (trans) $\frac{f^{(d_1)} \equiv g^{(d_2)} \quad g^{(d_2)} \equiv h^{(d_3)}}{f^{(d_1)} \equiv h^{(d_3)}}$

(wrfl) $\frac{f^{(d_1)} \sim g^{(d_2)}}{g^{(d_2)} \sim f^{(d_1)}}$ (wsym) $\frac{f^{(d_1)} \sim g^{(d_2)}}{g^{(d_2)} \sim f^{(d_1)}}$ (wtrans) $\frac{f^{(d_1)} \sim g^{(d_2)} \quad g^{(d_2)} \sim h^{(d_3)}}{f^{(d_1)} \sim h^{(d_3)}}$

(replsub) $\frac{f^{(d_1)} \equiv f^{(d_2)}}{g^{(d_2)} \circ f^{(d_1)} \equiv g^{(d_2)} \circ f^{(d_2)}}$

categorical rules

(comp) $\frac{f^{(d_1)} : X \rightarrow Y \quad g^{(d_1)} : Y \rightarrow Z}{(g \circ f)^{(d_1)} : X \rightarrow Z}$ (assoc) $\frac{f^{(d_1)} : X \rightarrow Y \quad g^{(d_2)} : Y \rightarrow Z \quad h^{(d_3)} : Z \rightarrow U}{h^{(d_3)} \circ (g^{(d_2)} \circ f^{(d_1)}) \equiv (h^{(d_3)} \circ g^{(d_2)}) \circ f^{(d_1)}}$

(ids) $\frac{f^{(d_1)} : X \rightarrow Y}{f^{(d_1)} \circ \text{id}_X^{(0)} \equiv f^{(d_1)}}$ (idt) $\frac{f^{(d_1)} : X \rightarrow Y}{\text{id}_Y^{(0)} \circ f^{(d_1)} \equiv f^{(d_1)}}$

(tcomp) $\frac{f^{(p)} : Y \rightarrow Z \quad g^{(p)} : X \rightarrow Y}{(\text{tpure} f)^{(0)} \circ (\text{tpure} g)^{(0)} \equiv (\text{tpure} (f \circ g))^{(0)}}$

Figure 11: $L_{\text{dec}}$: rules

Lemma 3.2. $\forall f^{(d_1)} : X \rightarrow Y, g^{(d_2)} : Y \rightarrow Z$, the annotation $(g \circ f)^{(\max(d_1,d_2))}$ is admissible.
**Proof:** Trivially follows from case analyses on $d_1$ and $d_2$, and the rules (0-to-1), (1-to-2) and (comp).

Hierarchically, a *pure* term can be seen as a *constructor* (0-to-1), and similarly a *constructor* term can be seen as a *modifier* on demand (1-to-2).

It is obviously free to convert strong equations into weak ones (stow). However, one has to make sure that the equated terms are not decorated with (2) in order to convert weak equations into strong ones with no further evidence (wtos).

Both strong and weak equalities are defined to be *equivalence relations* with the assumption that they are reflexive, transitive and symmetric. Strong equations form a congruence relation but weak equations do not: we will see this in detail when we specialize the decorated logic for the global state and the exception effects in Sections 4 and 5, respectively.

The categorical rules present properties of the term composition: the decoration of a composition depends on the decoration of its components, always taking the larger. I.e., $\forall f^{(0)}: X \to Y$ and $g^{(2)}: Y \to Z$, $g \circ f: X \to Z$ takes the decoration (2) (Lemma 3.2). Composition is an associative operation (assoc). The identity term disappears when to compose on the right (ids), and on the left (idt). The rule (tcomp) states that the $tpure$ constructor preserves the composition of pure terms up to the strong equality. Meaning that one can first compose pure terms outside the decorated environment (in any pure type system) and use the $tpure$ constructor to translate them into the $L_{dec}$, or translate the terms into the $L_{dec}$ first, and then compose them there. Notice that the red colored composition symbol (◦), in the rule conclusion, stands for the composition operation for pure terms. The decoration (p) of terms $f$ and $g$ is used just to denote the pure terms outside decorated environment, thus it does not take part in the decorated logic syntax. Similar case applies to the (tcomp) rule given in Figure 17.

### 4 The Decorated Logic for the state effect ($L_{st}$)

The use and modification of the memory state is the fundamental feature of imperative languages, and considered as a sort of computational side effect. In this section, we present a proof system for the use of the global state which involves access and modify operations, called the decorated logic for the state effect ($L_{st}$). This logic is obtained by extending the generic framework presented in Section 3.2. In this case, the decoration (0) is reserved for pure terms, while (1) is for read-only (accessor) and (2) is for read-write (modifier) terms. Two terms are called strongly equal if they return the same result with the same state manipulation; they are called weakly equal if they return the same result with different state manipulations.

Figure 12 shows the grammar of the $L_{st}$ where $1$ is the singleton type while $V_i$ is the type of values that can be stored in any location $i$. We assume that there is a set of locations called Loc. Given types $X$ and $Y$, we have $X \times Y$ representing type products.

Terms are closed under composition (◦) and pairing ($\langle \_, \_ \rangle_1$). I.e., for all terms $f: X \to Y$ and $g: Y \to Z$, we have $g \circ f: X \to Z$. Similarly, for all $f: X \to Y$ and $g: X \to Z$, there is $\langle f, g \rangle_1: X \to Y \times Z$. Notice that the pair subscript ‘1’ denotes the left pairs. One can define in a symmetric way the right pairs for terms $f: X \to Y$ and $g: X \to Z$ as $\langle f, g \rangle_2 := \text{permut} \circ \langle g, f \rangle_1$ where $\text{permut} := \langle \pi_2, \pi_1 \rangle_1$. In the same way, one can respectively obtain left and right products of terms $f: X_1 \to Y_1$ and $g: X_2 \to Y_2$ as $f \times_1 g := \langle f \circ \pi_1, g \circ \pi_2 \rangle_1$ and $f \times_2 g := \langle f \circ \pi_2, g \circ \pi_1 \rangle_2$. The term pairs/products are used to impose some order of term evaluation since the evaluation result depends on the order that the mutable state is accessed/modified. I.e., the product of two terms can be intuitively interpreted as they run on the global state in parallel, while sequential products, put forward in (Dumas et al., 2014a, §2.3), enforce terms to use the state in sequence. The decoration of a pair/product depends on the decoration of its components, always taking the larger. I.e., $\forall f^{(0)}: X \to Y$ and
where $\text{fst}$ a modifier $m$ follows:

$$
tpure \quad \text{translated from a pure type system with type products using the respect to the state effect.}
$$

$\text{lookup}$ only the terms including $\text{stored in a given location while}$ $\text{signatures close to their syntax and compose compatible terms as usual.}$

The term $\text{lookup}$ instance, $\text{contradictory assumptions.}$ See the constructor $\text{is_pair}$ of the inductive type $\text{is}.$

The intended model of the above grammar is built with respect to the set of states $S$ where a pure term $g^{(2)} : X \to Z,$ the term $\langle f, g \rangle_1 : X \to Y \times Z$ takes the decoration (2).

The interface terms are $\text{lookup},$ $\text{allowed to be constructed.}$ However, they cannot be used in the provided equational reasoning, since they may lead to conflicts on the returned result due to possible hazardous parallel modifications of the global state. We can have equational reasoning only when the left component is at most an accessor.

This restriction is given by the rules (w_lpair_eq) and (s_lpair_eq) in Figure 13. In the Coq implementation of this logic, as detailed in Section 4.2, we only allow the construction of pairs/products of modifiers under contradictory assumptions. See the constructor $\text{is_pair}$ of the inductive type $\text{is}.$

The interface terms are $\text{lookup},$ $\text{update}.$ We can call them $\text{accessor}$ while $\text{update}$ the unique sources of impurity $\text{accessor}$.

They are allowed to be constructed. However, they cannot be used in the provided equational reasoning, since they may lead to conflicts on the returned result due to possible hazardous parallel modifications of the global state. We can have equational reasoning only when the left component is at most an accessor. This restriction is given by the rules (w_lpair_eq) and (s_lpair_eq) in Figure 13. In the Coq implementation of this logic, as detailed in Section 4.2, we only allow the construction of pairs/products of modifiers under contradictory assumptions. See the constructor $\text{is_pair}$ of the inductive type $\text{is}.$

The intended model of the above grammar is built with respect to the set of states $S$ where a pure term $g^{(2)} : X \to Z,$ the term $\langle f, g \rangle_1 : X \to Y \times Z$ takes the decoration (2).

The interface terms are $\text{lookup},$ $\text{update}.$ We can call them $\text{accessor}$ while $\text{update}$ the unique sources of impurity $\text{accessor}$.

They are allowed to be constructed. However, they cannot be used in the provided equational reasoning, since they may lead to conflicts on the returned result due to possible hazardous parallel modifications of the global state. We can have equational reasoning only when the left component is at most an accessor. This restriction is given by the rules (w_lpair_eq) and (s_lpair_eq) in Figure 13. In the Coq implementation of this logic, as detailed in Section 4.2, we only allow the construction of pairs/products of modifiers under contradictory assumptions. See the constructor $\text{is_pair}$ of the inductive type $\text{is}.$

The intended model of the above grammar is built with respect to the set of states $S$ where a pure term $g^{(2)} : X \to Z,$ the term $\langle f, g \rangle_1 : X \to Y \times Z$ takes the decoration (2).

The intended model of the above grammar is built with respect to the set of states $S$ where a pure term $g^{(2)} : X \to Z,$ the term $\langle f, g \rangle_1 : X \to Y \times Z$ takes the decoration (2).
model is given in (Ekici, 2015, §5.1). The syntax given in Figure 12 is enriched with two sets of rules

Rules of the decorated logic for the state:

\[
\begin{align*}
\text{(pwepl)} \quad & \frac{\overline{f}^{(d_1)}_1 \sim \overline{f}^{(d_2)}_2 : X \to Y \ g : Y \to Z}{g^{(0)} \circ \overline{f}^{(d_1)}_1 \sim g^{(0)} \circ \overline{f}^{(d_2)}_2} \\
\text{(replsubs)} \quad & \frac{f^{(d_1)}_1 \equiv f^{(d_2)}_2 : X \to Y \ g^{(d_1)}_1 \equiv g^{(d_2)}_2 : Y \to Z}{g^{(d_1)}_1 \circ f^{(d_1)}_1 \equiv g^{(d_2)}_2 \circ f^{(d_2)}_2} \\
\text{(wsubs)} \quad & \frac{g^{(d_1)}_1 : X \to Y \ f^{(d_1)}_1 \sim f^{(d_2)}_2 : Y \to Z}{f^{(d_1)}_1 \circ g^{(d_3)}_1 \sim f^{(d_2)}_2 \circ g^{(d_3)}_2} \\
\text{(w_unit)} \quad & \frac{f^{(d_1)}_1 : X \to \emptyset}{f^{(d_1)}_1 \sim \langle \rangle^0} \\
\text{(ax₁)} \quad & \frac{\text{lookup}^{(1)}_1 \circ \text{update}^{(2)}_1 \sim \text{id}_{d_1}^{(0)}}{\text{lookup}^{(1)}_1 \circ \text{update}^{(2)}_1 \sim \text{lookup}^{(1)}_1 \circ \text{update}^{(2)}_1} \\
\text{(ax₂)} \quad & \frac{\forall i, j \in \text{Loc}, \ i \neq j}{\forall i, j \in \text{Loc}, \ i \neq j} \\
\text{(effect)} \quad & \frac{f^{(d_1)}_1, f^{(d_2)}_2 : X \to \emptyset}{\forall i \in \text{Loc}, \ \text{lookup}^{(1)}_1 \circ f^{(d_1)}_1 \sim \text{lookup}^{(1)}_1 \circ f^{(d_2)}_2} \\
\text{(local_global)} \quad & \frac{f^{(d_1)}_1, f^{(d_2)}_2 : X \to \emptyset \ \forall i \in \text{Loc}, \ \text{lookup}^{(1)}_1 \circ f^{(d_1)}_1 \sim \text{lookup}^{(1)}_1 \circ f^{(d_2)}_2}{f^{(d_1)}_1 \equiv f^{(d_2)}_2} \\
\text{(w_lpair_eq)} \quad & \frac{\pi^{(0)}_1 \circ (f^{(d_1)}_1, f^{(d_2)}_2) \sim f^{(d_1)}_1}{\pi^{(0)}_1 \circ (f^{(d_1)}_1, f^{(d_2)}_2 \max(d_1, d_2)) \sim f^{(d_1)}_1} \\
\text{(s_lpair_eq)} \quad & \frac{\pi^{(0)}_2 \circ (f^{(d_1)}_1, f^{(d_2)}_2) \equiv f^{(d_2)}_2}{\pi^{(0)}_2 \circ (f^{(d_1)}_1, f^{(d_2)}_2 \max(d_1, d_2)) \equiv f^{(d_2)}_2}
\end{align*}
\]

Figure 13: \( \mathcal{L}_R \): rules

presented in Figures 13 and 11. Weak equalities do not form a congruence: the term replacement cannot be done unless the replaced term is pure. I.e., given an equation \( f^{(d_1)}_1 \sim f^{(d_2)}_2 : X \to Y \) and a term \( g : Y \to Z \), it is possible to get the equation \( g \circ f^{(d_1)}_1 \sim g \circ f^{(d_2)}_2 \) only when the term \( g \) is pure. At this stage, we have no information about the modifications that \( f^{(d_1)}_1 \) and \( f^{(d_2)}_2 \) make on the memory state. Therefore, the post executed impure term \( g \) would destroy this result equality, for instance by reading the location \( i \) on which \( f^{(d_1)}_1 \) and \( f^{(d_2)}_2 \) have performed different modifications (pwepl). However, the term substitution can be done regardless of the term decoration. I.e., given the equation \( f^{(d_1)}_1 \sim f^{(d_2)}_2 : Y \to Z \) and a term \( g^{(d_3)}_1 : X \to Y \), it is possible to get the equation \( f^{(d_1)}_1 \circ g \sim f^{(d_2)}_2 \circ g \) independent from the decoration of the term \( g \). We already now that \( f^{(d_1)}_1 \) and \( f^{(d_2)}_2 \) return the same result, executing any term \( g \) in advance would not end them returning different results (wsubs). Strong equalities form a congruence by allowing both term substitutions and replacements independent from the term decorations (replsubs).

Any term \( f : X \to \emptyset \) with no result returned "\( \text{void} \)" (the unique inhabitant of \( \emptyset \) type) has an obvious result equality with the canonical empty pair \( \langle \rangle_\emptyset \) (w_unit).

The fundamental equations are given with the rules (ax₁) and (ax₂). The former states that by updating the location \( i \) with a value \( v \) and then observing the same location, one gets the value \( v \). This outputs the same value with the identity term \( \text{id}_{d_1} \), if it takes \( v \) as an argument. However, notice that these two ways of getting the value \( v \) have different state manipulations which makes them weakly equal. The latter, (ax₂), is to assume that updating the location \( j \) with a value \( v \) and then reading the content of a different location
Two modifiers \( f_1^{(2)}, f_2^{(2)} : X \to Y \) modify the state in the same way if and only if \( \langle \gamma \circ f_1 \rangle \equiv \langle \gamma \circ f_2 \rangle : X \to 1 \), where \( \langle \gamma \rangle : Y \to 1 \) throws out the returned value. So that \( f_1^{(2)}, f_2^{(2)} : X \to Y \) are strongly equal if and only if \( f_1 \sim f_2 \) and \( \langle \gamma \circ f_1 \rangle \equiv \langle \gamma \circ f_2 \rangle \) (effect). Notice that this rule is valid also for the other decorations of terms \( f_1 \) and \( f_2 \).

Locally, the strong equality between two modifiers \( f_1^{(2)}, f_2^{(2)} : X \to 1 \) can also be expressed as a pair of weak equations: \( f_1 \sim f_2 \) and \( \forall i : \text{Loc}, \text{lookup}_{i} \circ f_1 \sim \text{lookup}_{i} \circ f_2 \). The latter intuitively means that \( f_1 \) and \( f_2 \) leaves the memory with the same values stored in all (finitely many) locations after being executed. Given that both return “void” there is no explicitly need to check if \( f_1 \sim f_2 \). It suffices to see whether \( \forall i : \text{Loc}, \text{lookup}_{i} \circ f_1 \sim \text{lookup}_{i} \circ f_2 \) to end up with \( f_1 \equiv f_2 \) (local_global). The rule is valid also for the other decorations of terms \( f_1 \) and \( f_2 \).

With \( \langle \text{w_lpair_eq} \rangle \) and \( \langle \text{w_rpair_eq} \rangle \) term pairs are characterized: the (left) pair structure \( \langle f_1, f_2 \rangle_1 \) 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 \( \langle \text{w_lpair_eq} \rangle \) that \( \langle f_1, f_2 \rangle_1^{(\max(d_1, d_2))} \) has only result equality with \( f_1^{(d_1)} \) and by \( \langle \text{w_rpair_eq} \rangle \) that it has both result and effect equality with \( f_2^{(d_2)} \).

These rules are designed to be sound with respect to a categorical model detailed in (Ekici, 2015, §5.2, §5.3, §5.4, §5.5). However, their syntactic completeness is not immediate. Dumas et al. (2015) defines a new syntactic completeness property, subsuming a consistency check, called the relative Hilbert-Post completeness. In (Ekici, 2015, §5.4), it is proven that this set of rules is complete with due respect.

### 4.1 Decorated properties of the memory state

In (Plotkin and Power, 2002, §3), an equational representation of the mutable state has been introduced. The decorated version of such representation is given as follows:

1. (1) Annihilation lookup-update. Reading the content of a location \( i \) and then updating it with the obtained value is just like nothing. \( \forall i \in \text{Loc}, \text{update}_{i}^{(2)} \circ \text{lookup}_{i}^{(1)} \equiv \text{id}^{(0)}_{i} : 1 \to 1 \).

2. (2) Interaction lookup-lookup. Reading twice the same location \( i \) is the same as reading it once. \( \forall i \in \text{Loc}, \text{lookup}_{i}^{(1)} \circ \langle i \rangle_{0}^{1} \circ \text{lookup}_{i}^{(1)} \equiv \text{lookup}_{i}^{(1)} : 1 \to V_{i} \).

3. (3) Interaction update-update. Storing the values \( x \) and \( y \) in a row to the same location \( i \) is just like storing \( y \) in it. \( \forall i \in \text{Loc}, \text{update}_{i}^{(2)} \circ \pi_{2}^{(0)} \circ (\text{update}_{i}^{(2)} \times_{r} \text{id}_{V_{i}}^{(0)}) \equiv \text{update}_{i}^{(2)} \circ \pi_{2}^{(0)} : V_{i} \times V_{i} \to 1 \).

4. (4) Interaction update-lookup. Storing the value \( x \) in a location \( i \) and then reading the content of \( i \), one gets the value \( x \). \( \forall i \in \text{Loc}, \text{lookup}_{i}^{(1)} \circ \text{update}_{i}^{(2)} \sim \text{id}_{V_{i}}^{(0)} : V_{i} \to V_{i} \).

5. (5) Commutation lookup-lookup. The order of reading two different locations \( i \) and \( j \) does not matter. \( \forall i \neq j \in \text{Loc}, (\text{id}_{V_{i}}^{(0)} \times_{r} \text{lookup}_{j}^{(1)}) \circ \pi_{1}^{-1(0)} \circ \text{lookup}_{i}^{(1)} \equiv \text{permut}_{j,i}^{(0)} \circ (\text{id}_{V_{j}}^{(0)} \times_{r} \text{lookup}_{i}^{(1)}) \circ \pi_{1}^{-1(0)} \circ \text{lookup}_{j}^{(1)} : 1 \to V_{i} \times V_{j} \), where \( \pi_{1}^{-1(0)} := \langle \text{id}, \langle \rangle \rangle_{1}^{0} \).

6. (6) Commutation update-update. The order of storing in two different locations \( i \) and \( j \) does not matter. \( \forall i \neq j \in \text{Loc}, \text{update}_{j}^{(2)} \circ \pi_{2}^{(0)} \circ (\text{update}_{i}^{(2)} \times_{r} \text{id}_{V_{i}}^{(0)}) \equiv \text{update}_{i}^{(2)} \circ \pi_{1}^{(0)} \circ (\text{id}_{V_{i}}^{(0)} \times_{1} \text{update}_{j}^{(2)}) : V_{i} \times V_{j} \to 1 \).
We define the terms of $L$ we prefer a dependently typed implementation for higher readability. The constructor $\text{tpure}$ takes a Coq side (pure) function and translates it into the decorated environment. The constructor deals with the composition of two compatible terms. I.e., given a pair of terms $f: \text{term } X Y$ and $g: \text{term } Y Z$, then the composition $f \circ g$ would be an instance of the type $\text{term } X Z$. For the sake of conciseness, infix `$\circ$' is used to denote the term composition. Similarly, the (left) pair constructor is to constitute pairs of compatible terms. I.e., given $f: \text{term } X Y$ and $g: \text{term } Z X$, we have pair $(f, g)_1: \text{term } (Y \times X) X$. Instead of the symbol $(\_,\_)_1$, we use the keyword pair in the implementation. The terms $\text{lookup}$ and $\text{update}$ come as no surprise; just that the singleton type $\mathbb{I}$ and the type of values $V_i$ are respectively called $\text{unit}$ and $\text{Val } i$ in the code. The terms such as the identity, the pair projections, the singleton constant, and the constructor as follows:

\begin{align*}
&\text{Commutation } \text{update-lookup}. \text{ Just after storing a constant } c \text{ in a location } i, \text{ observing the content of } i \text{ is the same as regenerating the constant } c. \forall i \in \text{Loc}, \forall c \in V_i; \text{lookup}_i \circ \text{update}_i \circ \text{constant}_i \equiv \text{constant}_i \circ \text{update}_i \circ \text{constant}_i : \mathbb{I} \rightarrow V_i.
\end{align*}

These are the archetype properties that we have proved within the scope of the logic $L_{st}$. To see these proofs, check out author’s PhD thesis (Ekici, 2015, §5.3). Besides, we have implemented the $L_{st}$ in Coq to certify mentioned proofs. Section 4.2 details this implementation.

### 4.2 $L_{st}$ in Coq

In this section, we aim to highlight some crucial points of the $L_{st}$ implementation in Coq. It mainly consists of four steps: (1) implementing the terms, (2) assigning the decorations over terms, (3) stating the rules, and (4) proving properties of the memory state referred in Section 4.1.

We represent the set of memory locations by a Coq parameter $\text{Loc} : \text{Type}$. Since memory locations may contain different types of values, we also assume an arrow type $\text{Val} : \text{Loc} \rightarrow \text{Type}$ that is the type of values contained in each location. This fixes a type for every location. Note that the system thus does not support reasoning about strong updates.

```
Inductive term: Type -> Type -> Type ≜
| tpure: ∀ (X Y: Type), (X → Y) → term Y X
| 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
| lookup: ∀ i:Loc, term (Val i) unit
| update: ∀ i:Loc, term unit (Val i).
Infix "\_" ≜ comp (at level 70).
```

We define the terms of $L_{st}$ using an inductive predicate called $\text{term}$. It establishes a new Coq Type out of two input Types. The type $\text{term } X Y$ is dependent. It depends on the Type instances $X$ and $Y$, and represents the arrow type $X \rightarrow Y$ in the decorated framework. As opposed to a flat grammar with a typing predicate, we prefer a dependently typed implementation for higher readability.
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).

Remark that id is overloaded: defined one (on the left) is the identity of the decorated logic while the other one is the identity of Coq’s logic. The pair projections are named pi1 and pi2 while the unique mapping ⟨⟩\textsubscript{X} from any type X to 1 is named forget in the implementation.

The decorations are enumerated under the new type called kind: pure (0), ro (1) and rw (2) and inductively assigned to terms via the predicate called is. This predicate builds a proposition out of a term and a decoration. I.e., \forall i:Loc, is ro (lookup i) is a Prop instance, ensuring that “lookup i” is an accessor.

Notice that on the paper, we always mention the decoration of a term as a superscript. However, with such a Coq implementation, we do not need to additionally carry that information with a term. Instead, we inject it inside the rules as predicates, and check if a rule is applicable or not via this information. See Remark 4.1.

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

Any term that is built by the \texttt{tpure} constructor is pure (is\texttt{_tpure}). The decoration of any term composition depends on its components and always takes the upper decoration (pure < ro < rw). E.g., given a modifier term and a read-only term, their composition will be a modifier, as well. This trivially follows from (is\texttt{_comp}), (is\texttt{_pure_ro}) and (is\texttt{_ro_rw}): see Lemma 3.2, and the corresponding Coq proof here \textsuperscript{(iii)}. The decoration of a (left) pair of terms also depends on its components always taking the upper with the restriction that the first component can at most be an accessor. This is also trivial given (is\texttt{_pair}), (is\texttt{_pure_ro}) and (is\texttt{_ro_rw}). See the Coq proof of this fact here \textsuperscript{(iv)}. We declare that the term lookup is an accessor (is\texttt{_lookup}), and the term update is a modifier (is\texttt{_update}). The last two constructors (is\texttt{_pure_ro}) and (is\texttt{_ro_rw}) define the decoration hierarchies.

It is easy to derive that any \texttt{tpure} built term is pure. I.e., the purity proof of the first pair projection:

Lemma is\texttt{pi1} X Y: is pure (@pi1 X Y).
Proof. apply is\texttt{tpure}. Qed.

We now state the rules up to weak and strong equalities by defining them in a mutually inductive way: mutualivity here is used to enable the constructors including both weak and strong equalities. We use the

\textsuperscript{(iii)} \url{https://github.com/ekiciburak/decorated-logic-for-states-effect/blob/master/Decorations.v#L76-L79}
\textsuperscript{(iv)} \url{https://github.com/ekiciburak/decorated-logic-for-states-effect/blob/master/Decorations.v#L81-L84}
notation \(\equiv\) and \(\sim\) to denote strong and weak equalities, respectively.

\[
\text{Definition idem } X Y (x y: \text{term } X Y) \triangleq x = y.
\]

\[
\text{Inductive strong: } \forall X Y, \text{relation (term } X Y) \triangleq
\]

\[
\begin{align*}
& | \text{refl } X Y: \text{Reflexive (@strong } X Y) \\
& | \text{sym: } X Y, \text{Symmetric (@strong } X Y) \\
& | \text{trans: } X Y, \text{Transitive (@strong } X Y) \\
& | \text{replsubs: } X Y Z, \text{Proper (@strong } X Y \Rightarrow @strong Y Z \Rightarrow @strong X Z) \text{ comp} \\
& | \text{ids: } X Y (f: \text{term } X Y), \text{id} o f = f \\
& | \text{idt: } X Y (f: \text{term } X Y), \text{id} o f \equiv f \\
& | \text{assoc: } X Y Z T (f: \text{term } X Y), \text{is } f \rightarrow \text{is } g \rightarrow f \Rightarrow g = g \\
& | \text{wtoeq: } X Y (f g: \text{term } X Y), \text{is } f \Rightarrow \text{is } g \Rightarrow f \Rightarrow g = g \\
& | \text{s_lpair_eq: } X Y' Y (f1: \text{term } Y' Y) (f2: \text{term } Y' X), \text{is } f1 \rightarrow \text{pair } f1 f2 = f2 \\
& | \text{w_lpair_eq: } X Y' Y (f1: \text{term } Y' Y) (f2: \text{term } Y' X), \text{is } f1 \Rightarrow \text{pair } f1 f2 = f2 \\
& | \text{effect: } X Y (f g: \text{term } Y X), \text{forget } o f = \text{forget } o g \Rightarrow f \Rightarrow g = g \\
& | \text{local_global: } X (f g: \text{term } unit X), (\forall i: \text{Loc}, \text{lookup } i o f = \text{lookup } i o g) \Rightarrow f = g \\
& | \text{tcomp: } X Y Z (f: Z \Rightarrow Y) (g: Y \Rightarrow X), \text{tpure } (\text{compose } g f) = \text{tpure } g o \text{tpure } f \\
\end{align*}
\]

\[
\text{with weak: } X Y, \text{relation (term } X Y) \triangleq
\]

\[
\begin{align*}
& | \text{wsym: } X Y, \text{Symmetric (@weak } X Y) \\
& | \text{wtrans: } X Y, \text{Transitive (@weak } X Y) \\
& | \text{pwrep: } A B C (g: \text{term } C B), (\text{is pure } g) \Rightarrow \text{Proper } @\text{weak } B A \Rightarrow @\text{weak } C A \text{ (comp } g) \\
& | \text{wsubs: } A B C, \text{Proper } @\text{weak } B A \Rightarrow @\text{weak } C A \text{ (comp } g) \\
& | \text{wtoeq: } X Y Y (f1: \text{term } Y X) (f2: \text{term } Y' X), \text{is } f1 \Rightarrow \text{pair } f1 f2 = f1 \\
& | \text{w_lpair_eq: } X Y (f1: \text{term } Y X) (f2: \text{term } Y' X), \text{is } f1 \Rightarrow \text{pair } f1 f2 = f1 \\
& | \text{w_unit: } X (f g: \text{term } unit X), f = g \\
& | \text{ax1: } i, \text{lookup } i o \text{update } i = \text{id} \\
& | \text{ax2: } i j, i \neq j \Rightarrow \text{lookup } j o \text{update } i = \text{lookup } j o \text{forget} \\
\end{align*}
\]

where "\(x = y\)" \(\triangleq\) (strong \(x, y\)) and "\(x \sim y\)" \(\triangleq\) (weak \(x, y\)).

The rule tcomp states that the \(\text{tpure}\) constructor preserves the composition of pure terms up to the strong equality: one can first compose pure terms on Coq side (using higher order function \(\text{compose}\)) and then apply \(\text{tpure}\) constructor to translate them into decorated settings or can translate the terms first and then compose them in decorated settings.

**Remark 4.1.** In a decorated logic, it is crucial to verify the decorations of the terms in applying/rewriting a rule. If the rule is applicable for all decorations, then it is not necessary to check the decorations of terms which appear in that rule. Otherwise put, decoration checks are necessary only when the rule premise has restrictions over term decorations. I.e., see the constructor \(\text{w_lpair_eq}\) above. We apply the same strategy for the logics presented in Sections 5 and 6 when implementing them in Coq.

This framework allows us to express and prove, in Coq, the decorated versions of the properties mentioned in Section 4.1. E.g., the statement **commutation update-update** looks like:

\[
\begin{align*}
\text{Theorem CUU: } & \forall i j: \text{Loc}, i \neq j \Rightarrow \text{update } j o (\text{pi2 o (rprod (update i) (@id (Val j))))} = \\
& \text{update } i o (\text{pi1 o (lprod (@id (Val i)) (update j)))}. \\
\end{align*}
\]

where

\[
\begin{align*}
\text{Definition permut } (X Y): \text{term } (X*Y) (Y*X) & \triangleq \text{pair } \text{pi2 } \text{pi1}. \\
\text{Definition rpair } (X Y Z) (f: \text{term } X Y) (g: \text{term } Z X) & \triangleq \text{permut } o \text{pair } g f. \\
\text{Definition lprod } (X Y X' Y') (f: \text{term } X X') (g: \text{term } Y Y') & \triangleq \text{pair } (f o \text{pi1}) (g o \text{pi2}). \\
\text{Definition rprod } (X X' Y) (f: \text{term } X' X) (g: \text{term } Y Y') & \triangleq \text{permut } o \text{pair } (g o \text{pi2}) (f o \text{pi1}).
\end{align*}
\]
A fundamental feature of the exceptions mechanism is the distinction between ordinary values but differently on exceptional ones. The full Coq proofs of such properties can be found here (v), and the entire implementation there (vi).

5 The Decorated Logic for the exception effect ($\mathcal{L}_{\text{exc}}$)

Exception handling is provided by most modern programming languages to deal with anomalous or exceptional events which require special processing. In this section, we present a proof system for exceptions, which involves raising and handling operations, called the decorated logic for the exception effect ($\mathcal{L}_{\text{exc}}$). This logic is obtained by extending the generic framework presented in Section 3.2. In this context, the decoration (0) is reserved for pure terms, while (1) is for propagators and (2) is for catchers. A fundamental feature of the exceptions mechanism is the distinction between ordinary (non-exceptional) values and exceptions. Two terms are called strongly equal if they behave the same on ordinary and exceptional values; they are called weakly equal if they behave the same on ordinary values but differently on exceptional ones.

It has been shown by Dumas et al. (2012) that the core part of this proof system is dual to one for the state ($\mathcal{L}_s$). Based on this nice duality, we build the logic $\mathcal{L}_{\text{exc}}$, and detail it in the following.

**Grammar of the decorated logic for the exception:**

<table>
<thead>
<tr>
<th>(e ∈ EName)</th>
</tr>
</thead>
</table>

**Types:**

| t, s | ::= | X | Y | ... | t+s | ⊥ | EV_e |

**Decoration for terms:**

| (d_1), (d_2) | ::= | (0) | (1) | (2) |

**Terms:**

| f, g | ::= | a^d | b^d | ... | g ◦ f^d |

| [f^{d_1}: X → Y | g^{d_2}: Z → Y]_1^{\max(d_1,d_2)}: X + Z → Y |

| tag_e^1 | untag_e^2 | (↓ f)^{(1)} | (tpure e)^{(0)} |

**Equations:**

| eq | ::= | f^d | g^d | f^d ≈ g^d |

Figure 14 shows the grammar of $\mathcal{L}_{\text{exc}}$, where ⊥ is the empty (uninhabited) type while EV_e is the type of parameters for each exception name e. We assume that there is a finite set of exception names called EName. Given types X and Y, we have X + Y denoting co-product (disjoint union or sum) types. Terms are closed under composition (◦) and co-pairing ([_ | _]). I.e., for all terms f : X → Y and g : Y → Z, we have g ◦ f : X → Z. Similarly, for all f : X → Y and g : Z → Y, there is [f | g]_1 : X + Z → Y. Notice that the co-pair subscript ‘1’ denotes the left co-pairs. One can define in a symmetric way the right co-pairs for terms f : X → Y and g : Z → Y as [f | g]_r := [g, f]_r ◦ permut where permut := [1_n_2 | 1_n_1]. Similarly, one can respectively obtain left and right co-products (sums) of terms f : X_1 → Y_1 and g : X_2 → Y_2 as f + g := [1_n_1 ◦ f | 1_n_2 ◦ g]_r and f + r := [1_n_1 ◦ f | 1_n_2 ◦ g]_r. The decoration of a co-pair (co-product) depends on the decoration of its components, always taking the larger. I.e., ∀ f^{(0)} : X → Z and g^{(2)} : Y → Z, [f | g]_1 : X + Y → Z takes the decoration (2). Being dual to the pairs in $\mathcal{L}_s$ (which impose an evaluation order), co-pairs in $\mathcal{L}_{\text{exc}}$ are used to have case distinction among terms. Co-pairs of catchers are allowed to be constructed in the logic $\mathcal{L}_{\text{exc}}$. However, they cannot be used in the provided equational reasoning as they lead to ambiguous case distinctions over input exceptional arguments for the component terms. I.e., it is not obvious to which input argument the recovery would apply when both are exceptional. The intended

(v) https://github.com/ekiciburak/decorated-logic-for-states-effect/blob/master/Proofs.v

(vi) https://github.com/ekiciburak/decorated-logic-for-states-effect
equational reasoning can be done only when the left term is at most a propagator. The restriction is given by the rules (w_lcopair_eq) and (s_lcopair_eq) in Figure 15.

The interface terms are $\text{tag}_e : E_V \rightarrow \mathbb{O} + E$ and $\text{untag}_e : \mathbb{O} + E \rightarrow E_V + E$ where $E$ denotes the distinguished object of exceptions which never appears in the decorated setting. The use of decorations provides a new schema where term signatures are constructed without any occurrence of it. For instance, $\text{tag}_e^{(1)} : E_V \rightarrow \mathbb{O}$ is a thrower while $\text{untag}_e^{(2)} : \mathbb{O} \rightarrow E_V$ is a catcher. This way, we keep signatures close to their syntax and compose compatible terms as usual. The term $\text{tag}_e$ encapsulates an ordinary value with an exception of name $e$ while the term $\text{untag}_e$ recovers the value from the exceptional case.

The ‘↓’ symbol denotes the downcast term that takes as input a term and prevents it from catching exceptions. It is used when to define the try/catch block in this setting. See Definition 5.2.

The identity term $\text{id}$, the canonical co-pair inclusions $\text{in}_1$ and $\text{in}_2$, and the empty co-pair $\mathbb{O}$ (used to convert the type of input exceptional value into the given type; $X$ in this case) are translated from a pure type system with sum types using the $\text{tpure}$ constructor, for all types $X$ and $Y$, as follows:

\[
\begin{align*}
\text{id}_X^{(0)} & : X \rightarrow X := \text{tpure} \ (\lambda x : X. x : X) \\
\text{in}_1^{(0)} & : X \rightarrow X + Y := \text{tpure} \ \text{inl} \\
\text{in}_2^{(0)} & : Y \rightarrow X + Y := \text{tpure} \ \text{inr} \\
[\mathbb{O}]_X^{(0)} & : \mathbb{O} \rightarrow X := \text{tpure} \ (\lambda \_ : \mathbb{O}. x : X)
\end{align*}
\]

where $\text{inl}$ and $\text{inr}$ are constructors of sum types, and in the definition of $[\mathbb{O}]_X$, $X$ is assumed to be inhabited.

The intended model of the grammar of the logic $\mathcal{L}_{\text{exc}}$ is built with respect to the set of exceptions $E$ where a pure term $p^{(0)} : X \rightarrow Y$ is interpreted as a function $p : X \rightarrow Y$, a propagator $p p^{(1)} : X \rightarrow Y$ as a function $p p : X \rightarrow Y + E$, and a catcher $c^{(2)} : X \rightarrow Y$ as a function $c : X + E \rightarrow Y + E$. The complete and detailed category theoretical model is given in (Ekici, 2015, §6.1).

**Definition 5.1.** For each type $Y$ and exception name $e$, the propagator $\text{throw}_{Y,e}^{(1)}$ is defined as:

\[
\text{throw}_{Y,e}^{(1)} := [\mathbb{O}]_Y \circ \text{tag}_e^{(1)} : E_V \rightarrow Y
\]

Intuitively, raising an exception of name $e$ is first tagging the given ordinary value with $e$ and then coercing the empty type into $Y$ for the continuation issues.

**Definition 5.2.** For each propagators $f^{(1)} : X \rightarrow Y$, $g^{(1)} : E_V \rightarrow Y$ and each exception name $e$, the propagator $\text{try}(f)\text{catch}(e \Rightarrow g)^{(1)}$ is defined in three steps, as follows:

\[
\begin{align*}
\text{Catch}(e \Rightarrow g)^{(2)} & := [\mathbb{O}]_Y \circ \text{id}_X^{(0)} \circ \text{untag}_e^{(2)} \ : Y + \mathbb{O} \rightarrow Y \\
\text{Try}(f)\text{Catch}(e \Rightarrow g)^{(2)} & := \text{Catch}(e \Rightarrow g)^{(2)} \circ \text{in}_1^{(0)} \circ f^{(1)} \ : X \rightarrow Y \\
\text{try}(f)\text{catch}(e \Rightarrow g)^{(1)} & := \downarrow(\text{Try}(f)\text{Catch}(e \Rightarrow g)^{(2)}) \ : X \rightarrow Y
\end{align*}
\]

To handle an exception, the intermediate expressions $\text{Catch}(e \Rightarrow g)$ and $\text{Try}(f)\text{Catch}(e \Rightarrow g)$ are private catchers and the expression $\text{try}(f)\text{catch}(e \Rightarrow g)$ is a public propagator: the downcast operator intuitively used to prevent $\text{try}(f)\text{catch}(e \Rightarrow g)$ from catching exceptions with name $e$ which might have been raised.
before its execution. Below we depict the \(\text{try}(f)\text{catch}(e \Rightarrow g)\) definition as a diagram:

\[
\begin{align*}
\downarrow (X \xrightarrow{f} Y) & \quad \xrightarrow{\text{id}_Y, O} \quad Y + \O \quad \xrightarrow{\text{untag}_e} \quad Y \\
\downarrow & \quad \xrightarrow{\text{id}_Y, [g \circ \text{untag}_e]} \quad Y \\
\O & \quad \xrightarrow{\text{untag}_e} \quad EV_e
\end{align*}
\]

This, inside the downcast, intuitively tells us that if the term \(f\) throws an exception, then within the Catch block using the case distinction, provided by the copair, the exception is handled via the \(\text{untag}\) (unhandled exception gets propagated) and the continuation is the execution of the term \(g\). If \(f\) does not throw any exception then no handling is performed, we have \(\text{id}\) term in execution. Note also that the term \(\text{in}_{\text{Y}, O}\) used in the Definition 5.2 is the horizontal one in the above diagram, and implicitly means that \(Y\) and \(Y + \O\) isomorphic objects. The inclusions \(\text{in}_{\text{Y}, O}\) (the vertical in the above diagram) and \(\text{in}_{\text{2Y}, O}\) play a role in the equational reasoning, given in Figure 15, that we provide on the top of \(Z_{\text{exc}}\) syntax.

The definition of \(\text{try}(f)\text{catch}(e \Rightarrow g)\) corresponds to the Java mechanism for exceptions as in (Gosling et al., 2005, §14) and in (Jacobs (2001)) with the following control flow (where \(\text{exc}\) means “is this value an exception?”): an \textit{abrupt} termination returns an uncaught exception and a \textit{normal} termination returns an ordinary value.

Remark 5.3. The decorated terms \(\text{throw}^{(1)}\) and \(\text{throw}/\text{catch}^{(1)}\) stated in Definitions 5.1 and 5.2 will serve, in Section 7 (see the translator function \(d\text{Cmd}\)), as interpretations of the \(\text{IMP+Exc}\) commands \(\text{THROW}\) and \(\text{TRY/CATCH}\).

The syntax given in Figure 14 is enriched with two sets of rules presented in Figures 15 and 11. Weak equalities do not form a congruence: the term substitution cannot be done unless the substituted term is pure. I.e., given the equation \(f_1^{(d_1)} \sim f_2^{(d_2)} : Y \rightarrow Z\) and a term \(g : X \rightarrow Y\), it is possible to get the equation
Rules of the decorated logic for the exception:

\[
\begin{align*}
\text{(wrepl)} & \quad f^{(d_1)} \circ g \sim f^{(d_2)} \circ g \\
\text{(replsubs)} & \quad f^{(d_1)} \circ f^{(d_2)} : X \rightarrow Y \\
\text{(w_downcast)} & \quad f^{(2)} : Y \rightarrow X \\
\text{(e1x)} & \quad f^{(d_1)} \circ f^{(d_2)} : Y \rightarrow X \\
\text{(eefect)} & \quad f^{(d_1)} \circ f^{(d_2)} : Y \rightarrow X \\
\text{(elocalt_global)} & \quad f^{(d_1)} \circ f^{(d_2)} : Y \rightarrow X \\
\text{(w_lcopair_eq)} & \quad f^{(d_1)} \circ f^{(d_2)} : Z \rightarrow Y \\
\text{(s_lcopair_eq)} & \quad f^{(d_1)} \circ f^{(d_2)} : Z \rightarrow Y
\end{align*}
\]

\( f_1 \circ g \sim f_2 \circ g \) only when the term \( g \) is pure. At this stage, we have no information about the behaviors of \( f_1 \) and \( f_2 \) on exceptional values. Therefore, the pre-executed term \( g \) would destroy this result equality unless being pure, for instance, by throwing an exception of name \( e \) for which \( f_1 \) and \( f_2 \) perform different behaviors: say one is propagating, while the other is recovering from it (pwsubs). However, the term replacement can be done regardless of the term decoration. I.e., given the equation \( f^{(d_1)} \sim f^{(d_2)} : X \rightarrow Y \) and a term \( g^{(d_3)} : Y \rightarrow Z \), it is possible to get the equation \( g \circ f_1 \sim g \circ f_2 \) independent from the decoration of the term \( g \). Since \( f_1 \) and \( f_2 \) behave the same on ordinary values, executing any term \( g \) after \( f_1 \) and \( f_2 \) would not end them behave different on ordinary values (wrepl). Strong equalities form a congruence by allowing both term substitutions and replacements regardless of the term decorations (replsubs).

Any term \( f : \emptyset \rightarrow X \) with no input parameter has an equivalence on ordinary values with the empty co-pair \( \emptyset \) (w_empty). The rule (w_downcast) states that the term \( \downarrow f \) behaves as \( f \), if the argument is ordinary. The fundamental equations are given with the rules (e1x1) and (e1x2). The former states that encapsulating an ordinary value with an exception of name \( e \) followed by an immediate recovery would be equivalent to “doing nothing” in terms of ordinary values. Clearly, this is only a weak equation since its sides behave different on exceptional values: left hand side may recover but right hand side definitely propagates. The latter, (e1x2), is to assume that encapsulating an ordinary value \( v \) with an exception of name \( e_2 \) and then trying to recover it from a different exception of name \( e_1 \) would just lead \( e_2 \) to be propagated. Similarly, if the ordinary value \( v \) is encapsulated with \( e_2 \) with no recovery attempt afterwards.
would again lead \( e_2 \) to be propagated. These two operations behave the same on ordinary values but different on exceptional ones. For instance, left hand side recovers the input value (encapsulated with the exception name \( e_1 \)) while right hand side propagates it.

Two catchers \( f_1^{(2)}, f_2^{(2)} : X \to Y \) behave the same on exceptional values if and only if \( f_1 \circ [ ]_X \equiv f_2 \circ [ ]_X \), where \( [ ]_X : \emptyset \to X \) throws out exceptional values. So that \( f_1^{(2)}, f_2^{(2)} : X \to Y \) are strongly equal if and only if \( f_1 \sim f_2 \) and \( f_1 \circ [ ]_X \equiv f_2 \circ [ ]_X \) (effect). The rule is valid also for the other decorations of terms \( f_1 \) and \( f_2 \).

Strong equality between two catchers \( f_1^{(2)}, f_2^{(2)} : \emptyset \to X \) can also be expressed as a pair of weak equations: \( f_1 \sim f_2 \) and \( \forall e : \text{EName}, f_1 \circ \text{tag}_e \sim f_2 \circ \text{tag}_e \). The latter intuitively means that \( f_1 \) and \( f_2 \) behaves the same on all (finitely many) exceptional values when executed. Given that both behave the same on ordinary arguments (due to \( \text{w\_empty} \)), there is no explicitly need to check if \( f_1 \sim f_2 \). It suffices to see whether \( \forall e : \text{EName}, f_1 \circ \text{tag}_e \sim f_2 \circ \text{tag}_e \) to end up with \( f_1 \equiv f_2 \) (\text{elocal\_global}). This rule is valid also for the other decorations of terms \( f_1 \) and \( f_2 \).

With \( \text{w\_copair\_eq} \) and \( \text{w\_rcopair\_eq} \), term co-pairs (sums) are characterized: the (left) co-pair structure \( [f_1 | f_2]_1 \) cannot be used when \( f_1 \) and \( f_2 \) both are catchers, since it may lead to a conflict on exceptional values. When \( f_1 \) is a propagator, with \( \text{w\_copair\_eq} \), we assume that ordinary values of type \( X \) are treated by \( [f_1 | f_2]_1^{(\max(d_1,d_2))} \) as they would be by \( f_1^{(d_1)} \) and with \( \text{w\_copair\_eq} \) that ordinary values of type \( Y \) and exceptional values are treated by \( [f_1 | f_2]_2^{(\max(d_1,d_2))} \) as they would be by \( f_2^{(d_2)} \).

Similar to the ones of the logic \( \mathcal{L}_d \), the rules of the logic \( \mathcal{L}_{\text{exc}} \) also designed to be sound with respect to a categorical model which is detailed in (Ekici, 2015, §6.2, §6.3, §6.4, §6.5). In (Dumas et al. (2015)), we prove that this set of rules is complete with respect to the notion of relative Hilbert-Post completeness.

### 5.1 Decorated properties of the exception effect

Similar to the one for the state effect presented in Section 4.1, we propose an equational representation of the exception effect with the following decorated equations:

1. **Annihilation tag-untag.** Untagging an exception of name \( e \) and then raising it again is just like doing nothing. \( \forall e \in \text{EName}, \text{tag}_e^{(1)} \circ \text{untag}_e^{(2)} \equiv \text{id}_X^{(0)} : \emptyset \to \emptyset. \)

2. **Commutation untag-untag.** Untagging two distinct exception names can be done in any order.
   \[ \forall e \neq r \in \text{EName}, \ (\text{untag}_e + \text{id}_{\text{Ev}_e})^{(2)} \circ \text{untag}_r^{(2)} \equiv \text{id}_{\text{Ev}_e + \text{Ev}_r} \circ \text{untag}_e^{(2)} \circ \text{untag}_r^{(2)} : \emptyset \to \text{Ev}_e + \text{Ev}_r. \]

3. **Propagator-propagates.** A propagator term always propagates the exception.
   \( \forall e \in \text{EName}, \text{a}^{(1)} : X \to Y, \ a^{(1)} \circ [ ]_Y \circ \text{tag}_e^{(1)} \equiv [ ]_Y \circ \text{tag}_e^{(1)} : \text{Ev}_e \to Y. \)

4. **Recovery.** The parameter used for throwing an exception may be recovered.
   \[ (\forall f^{(1)}, g^{(1)} : X \to \emptyset, \ [ ]_Y \circ f^{(1)} \equiv [ ]_Y \circ g^{(1)} \implies f^{(1)} \equiv g^{(1)} ) \implies \]
   \[ (\forall e \in \text{EName}, \ u_1^{(0)} : X \to \text{Ev}_e, \ (\text{throw}_e^{(1)} \circ u_1^{(0)} \equiv \text{throw}_e^{(1)} \circ u_2^{(0)} ) \implies u_1^{(0)} \equiv u_2^{(0)} : X \to \text{Ev}_e). \]

5. **Try.** The strong equation is compatible with \text{try/catch}.
   \[ \forall e \in \text{EName}, \text{a}^{(1)}_1, \text{a}^{(1)}_2 : X \to Y, \text{b}^{(1)} : \text{Ev}_e \to Y, \text{a}^{(1)}_1 \equiv \text{a}^{(1)}_2 \implies \text{try}(\text{a}_1)\text{catch}(e \Rightarrow \text{b}^{(1)}) \equiv \text{try}(\text{a}_2)\text{catch}(e \Rightarrow \text{b}^{(1)} : X \to Y. \)
(6)_d Try0. Pure code inside try never triggers the code inside catch.
∀ e ∈ EName, u(0) : X → Y, b(1) : EV_e → Y, try(u) / catch(e ⇒ b)(1) ≡ u(0) : X → Y.

(7)_d Try1. The code inside catch is executed as soon as an exception is thrown inside try.
∀ e ∈ EName, u(0) : X → EV_e, b(1) : EV_e → Y, try(throw_e ◦ u) / catch(e ⇒ b)(1) ≡ b(1) ◦ u(0) : X → Y.

(8)_d Try2. An exception gets propagated, if the exception name is not pattern matched in catch.
∀ (e ≠ f) ∈ EName, u(0) : X → EV_f, b(1) : EV_e → Y, try(throw_f ◦ u) / catch(e ⇒ b)(1) ≡ throw_f ◦ u(0) : X → Y.

These are the archetype properties that we have proved within the scope of the \( \mathcal{L}_{\text{exc}} \). To see these proofs, check out (Ekici, 2015, §6.7). Besides, we have implemented the \( \mathcal{L}_{\text{exc}} \) in Coq to certify mentioned proofs. Section 5.2 briefly discusses this implementation. Notice that the premise of the property (4)_d is a very specific mono requirement. It intuitively says that if there is a strong equality between two propagators (i.e., \( f(1) \) and \( g(1) \)) after removing the exceptional values they may propagate, then they are strongly equal. In the absence of this requirement the property is not valid.

5.2 \( \mathcal{L}_{\text{exc}} \) in Coq

Coq implementation of \( \mathcal{L}_{\text{exc}} \) follows the same approach with the one for \( \mathcal{L}_{\text{st}} \) as summarized in Section 4.2. We represent the set of exception names by a Coq parameter \( \text{EName} : \text{Type} \). An arrow type \( \text{EVal} : \text{EName} \to \text{Type} \) is assumed as the type of values (parameters) for each exception name. We then inductively define terms and assign decorations over them. There, we respectively use keywords \( \text{epure} \), \( \text{ppg} \) and \( \text{ctc} \) instead of (0), (1) and (2). The rules up to weak and strong equalities are stated in a mutually inductive way to allow constructors including both types of equalities, similar to the approach presented in Section 4.2. We choose not to replay the entire Coq encoding here, but at least give Coq formalizations of Definitions 5.1 and 5.2:

The encodings of other terms are contained in this file \(^{(vii)}\).

We can conclude that such a framework allows us to express and prove, in Coq, the decorated versions of the properties mentioned in Section 5.1. E.g., the statement propagator-propagates looks like:

\[ (\text{Propagator propagates}) \]
Lemma PPT: \( \forall X \ Y \ (e : \text{EName}) \ (a : \text{term Y X}), \text{is ppg a} \to a @ (\text{empty Y} @ \text{tag e}) = (\text{empty Y} @ \text{tag e}). \]

The full Coq proofs of such properties can be found here \(^{(viii)}\), and the entire implementation there \(^{(ix)}\).

6 Combining \( \mathcal{L}_{\text{st}} \) and \( \mathcal{L}_{\text{exc}} \)

In order to formally cope with different computational effects, one needs to compose the related formal models. For instance, using monad transformers (Jaskelioff (2009)), it is usually possible to combine

\(^{(vii)}\) https://github.com/ekiciburak/decorated-logics-for-exceptions-effect/blob/master/Terms.v

\(^{(viii)}\) https://github.com/ekiciburak/decorated-logics-for-exceptions-effect/blob/master/Proofs.v

\(^{(ix)}\) https://github.com/ekiciburak/decorated-logics-for-exceptions-effect
effects formalized by monads, as encoded in Haskell. Handler compositions allow combining effects modeled by algebraic handlers, as implemented in Eff by Bauer and Pretnar (2015, 2014); Pretnar (2014) and in Idris by Brady (2013). To combine effects formalized in decorated settings, we just need to compose the related logics. In this section, we formally study the combination of the state and the exception effects using the logics $L_{st}$ and $L_{exc}$. We call the newly born logic the decorated logic for the state and the exception, and denote it $L_{st+exc}$. To start with, we give the syntax of $L_{st+exc}$ below in Figure 16.

**Grammar of the decorated logic for the state and the exception:**

(i $\in$ Loc)  (e $\in$ EName)

**Grammar of the decorated logic for the state and the exception:**

| Types: | t, s | ::= X | Y | $\cdots$ | t x s | t+s | $\emptyset$ | $\top$ | $\bigvee$ | $V_i$ | EV_s |
| Terms: | f, g | ::= a^{(d_1,d_2)} | b^{(d_1,d_2)} | $\cdots$ | g $\circ$ f^{(d_1,d_2)} | $\langle f^{(d_1,d_2)}, g^{(d_1,d_2)} \rangle_{\max(d_1,d_3), \max(d_2,d_4)}} | $\langle f^{(d_1,d_2)}, g^{(d_2,d_4)} \rangle_{\max(d_1,d_3), \max(d_2,d_4)}} | $\lambda$ f^{(d_1,d_2)} | update_{\{0\}} | tag_{\{0\}} | untag_{\{0\}} | (\downarrow f)^{(0,1)} | (tpure^\ast)^{(0,0)} |

**Equations:**
eq ::= f^{(d_1,d_2)} \equiv g^{(d_1,d_2)} | f^{(d_1,d_2)} \sim \equiv g^{(d_1,d_2)} | f^{(d_1,d_2)} \sim \sim g^{(d_1,d_2)} |

Figure 16: $L_{st+exc}$: syntax

The decorations are paired off to cover all possible combinations: the decoration symbol on the left is given in terms of the state effect while the one on the right is of the exception. I.e., $f^{(1,2)}$ says that $f$ may access to the state alongside catching exceptions. The decoration of a (co)-pair(co)-product or a composition depends on the decorations of its components, always taking the larger. I.e., $\forall f^{(1,2)} : X \rightarrow Y$ and $g^{(2,1)} : Y \rightarrow Z$, $g \circ f : X \rightarrow Z$ takes the decoration $(2, 2)$. The pairs/products of compatible terms $f_1^{(2,2)}$, $g_1^{(2,2)}$, and similarly the co-pair/co-products of compatible terms $f_2^{(2,2)}$, $g_2^{(2,2)}$ can be constructed within the scope of $L_{st+exc}$ but cannot be used in the provided equational reasoning. This is because, $f_1^{(2,2)}$ and $g_1^{(2,2)}$, as two modifiers, may lead to conflicts on the returned results over any type of (exceptional or ordinary) arguments due to the possible hazardous parallel modifications of the global state, while $f_2^{(2,2)}$ and $g_2^{(2,2)}$, as two catchers, may yield in ambiguous case distinctions over input exceptional arguments. I.e., it is not obvious to which input argument the recovery would apply when both are exceptional. The rules (w_lonpair_eq), (s_lonpair_eq), (w_locpair_eq) and (s_locpair_eq), in Figure 17, enforce these restrictions.

The types of $L_{st+exc}$ is the union of the types of $L_{st}$ and $L_{exc}$. Similarly, the terms of $L_{st+exc}$ is the union of the terms of $L_{st}$ and $L_{exc}$. The interface terms for the state effect are pure with respect to the exception and vice versa: lookup^{(1,0)}, update^{(2,0)}, tag^{(0,1)} and untag^{(0,2)}. As in Sections 4 and 5, we use the special tpure constructor to translate pure terms such as the identity id, the canonical pair projections $\pi_1$, $\pi_2$, the empty pair $\langle \rangle$, the canonical co-pair inclusions $1_1$, $1_2$, the empty co-pair $\langle \rangle$ and constants from a pure type system with product and sum types using the tpure constructor, for all types $X$ and $Y$, as:

$id_{X}^{(0,0)} : X \rightarrow X \quad := \quad \text{tpure} \ (\lambda x : X . x : X)$

$\pi_{1}^{(0,0)} : X \times Y \rightarrow X \quad := \quad \text{tpure} \ fst$
We plan it as a future work to come up with a more general and systematic way to combine effects formalized within decorated logics.

\[
\begin{align*}
\pi_2^{(0,0)} &: X \times Y \to Y := \text{tpure} \text{ and} \\
\langle \cdot \rangle^{(0,0)} &: X \to \mathbb{I} := \text{tpure} (\lambda x : X. \text{void} : \mathbb{I}) \\
i_{n_1}^{(0,0)} &: X \to X + Y := \text{tpure} \text{ inl} \\
i_{n_2}^{(0,0)} &: Y \to X + Y := \text{tpure} \text{ inr} \\
[\cdot]^{(0,0)} &: \mathbb{O} \to X := \text{tpure} (\lambda _{-} : \mathbb{O}. x : X)
\end{align*}
\]

where \text{fst} and \text{snd} are constructors of product types while \text{inl} and \text{inr} are of sum types, and in the definition of \([\cdot]_X, X\) is assumed to be inhabited.

The rule combinations need a bit of reformulation as we summarize below:

- The decoration symbol \((0)\) freely converts into \((1)\) and \((2)\), while the symbol \((1)\) just into \((2)\) when the other symbol is fixed. I.e., \(f^{(0,2)}\) freely converts into \(f^{(1,2)}\). See all cases below:

  \[
  \frac{f^{(0,d)}}{f^{(1,d)}}, \quad \frac{f^{(1,d)}}{f^{(2,d)}}, \quad \frac{f^{(d,0)}}{f^{(d,1)}}, \quad \frac{f^{(d,1)}}{f^{(d,2)}} \quad \text{for } d \in \{0, 1, 2\}
  \]

- We have all possible combinations of equality sorts: \(\equiv\equiv, \equiv\sim, \sim\equiv\) and \(\sim\sim\). The first equality symbol relates terms with respect to the state effect. I.e., \(f \equiv g\) means that \(f\) and \(g\) are strongly equal with respect to the state, while being weakly equal with respect to the exception. Below we present the conversion rules between these four sorts. The burden here is that a strong equality symbol can always be freely converted into a weak one independent of according to which effect it relates terms. But, to convert a weak equality symbol into a strong one, we need to make sure that the related terms are decorated either with \((0)\) or \((1)\) with respect to the effect they are weakly related.

  \[
  \begin{align*}
  \frac{f^{(2,2)}}{f^{(2,2)}}, \quad \frac{f^{(2,2)}}{f^{(2,2)}}, \quad \frac{f^{(2,2)}}{f^{(2,2)}}, \quad \frac{f^{(2,2)}}{f^{(2,2)}}
  \end{align*}
  \]

- The rules of the logic \(L_{at+exec}\) are presented in Figure 17 as a union of the ones given in Figures 13 and 15 in terms of new equality sorts and refined term decorations. There, we reprise the whole rule bodies, and implicitly assume that all equality sorts are equivalence relations respecting the properties \text{reflexivity, symmetry, and transitivity}.

We plan it as a future work to come up with a more general and systematic way to combine effects formalized within decorated logics.
Rules of the decorated logic for the state and the exception:

- (assoc) $\forall (d_1, d_2) : X \to Y \quad g^{(d_2, d_1)} : Y \to Z \quad h^{(d_1, d_2)} : Z \to T$
  \[
  f_{(d_1, d_2)} : X \to Y \quad g^{(d_2, d_1)} \circ (f_{(d_1, d_2)}) \equiv (h^{(d_1, d_2)} \circ g^{(d_2, d_1)}) \circ f_{(d_1, d_2)}
  \]

- (ids) $\forall (d_1, d_2) : X \to Y$
  \[
  f_{(d_1, d_2)} \circ id_0^{(d_2, d_1)} \equiv id_0^{(d_2, d_1)} \circ f_{(d_1, d_2)} \equiv f_{(d_1, d_2)}
  \]

- (pwrepl) $\forall (d_1, d_2) : X \to Y \quad f_{(d_1, d_2)} \equiv f_{(d_2, d_1)} : X \to Y \quad g^{(d_2, d_1)} : Y \to Z$
  \[
  f_{(d_1, d_2)} \circ g^{(d_2, d_1)} \equiv g^{(d_2, d_1)} \circ f_{(d_1, d_2)}
  \]

- (pwsbs) $\forall (d_1, d_2) : X \to Y \quad f_{(d_1, d_2)} \equiv f_{(d_2, d_1)} : X \to Y \quad g^{(d_2, d_1)} : Y \to Z$
  \[
  f_{(d_1, d_2)} \circ g^{(d_2, d_1)} \equiv g^{(d_2, d_1)} \circ f_{(d_1, d_2)}
  \]

- (replbs) $\forall (d_1, d_2) : X \to Y \quad g^{(d_2, d_1)} : Y \to Z$
  \[
  f_{(d_1, d_2)} \equiv f_{(d_2, d_1)} : X \to Y \quad g^{(d_2, d_1)} \circ f_{(d_1, d_2)} \equiv g^{(d_2, d_1)} \circ f_{(d_2, d_1)}
  \]

- (w_unit) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to X$
  \[
  f_{(d_1, d_2)} : X \to \alpha \equiv \alpha \circ f_{(d_1, d_2)}
  \]

- (w_empty) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to X$
  \[
  f_{(d_1, d_2)} \equiv \alpha \circ f_{(d_1, d_2)}
  \]

- (local_global) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to X$
  \[
  f_{(d_1, d_2)} \equiv f_{(d_2, d_1)} : X \to \alpha
  \]

- (w_lpair) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to Z$
  \[
  f_{(d_1, d_2)} : X \to \alpha \equiv \alpha \circ f_{(d_1, d_2)}
  \]

- (s_lpair) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to Z$
  \[
  f_{(d_1, d_2)} : X \to \alpha \equiv \alpha \circ f_{(d_1, d_2)}
  \]

- (w_lcopair) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to Z$
  \[
  f_{(d_1, d_2)} : X \to \alpha \equiv \alpha \circ f_{(d_1, d_2)}
  \]

- (s_lcopair) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to Z$
  \[
  f_{(d_1, d_2)} : X \to \alpha \equiv \alpha \circ f_{(d_1, d_2)}
  \]

- (tcomp) $\forall (d_1, d_2) : X \to Y \quad \forall \alpha : Y \to Z$
  \[
  f_{(d_1, d_2)} : X \to \alpha \equiv \alpha \circ f_{(d_1, d_2)}
  \]

Figure 17: $L_{\mu, exc}$: rules
6.1 Decorated properties of the state and exception effects

The properties given in Sections 4.1 and 5.1 are now stated with the refined term decorations, and related with the equation sort \( \equiv \). I.e., the statements propagator-propagates and update-update look like:

\[
\forall e \in \text{EName}, a^{(0,1)}: X \rightarrow Y, a^{(0,1)} \circ [X^{(0,0)} \circ \text{tag}_e^{(0,1)}] \equiv [Y^{(0,0)} \circ \text{tag}_e^{(0,1)}]: \text{EV}_e \rightarrow Y.
\]

\[
\forall i \neq j \in \text{Loc}, \text{update}_{i}^{(2,0)} \circ \pi_2^{(0,0)} \circ (\text{update}_{i}^{(2,0)} \times_1 \text{id}_{j}^{(0,0)}) \equiv \\
\text{update}_{i}^{(2,0)} \circ \pi_2^{(0,0)} \circ (\text{id}_{i}^{(0,0)} \times_1 \text{update}_{j}^{(2,0)}): V_i \times V_j \rightarrow I.
\]

These are the archetype properties that we can prove within the scope of the \( \mathcal{L}_{st+exc} \). Although it is doable, we prefer not to prove them for this generic framework (skipped since it would take substantial amount of time); instead, we first specialize them in a way to serve as a target language for a denotational semantics of \( \text{IMP}+\text{Exc} \), and then prove them for the specialized version. Also, we encode the specialized version in Coq and certify related proofs. Section 7 gives the related details.

7 IMP+Exc over the combined decorated logic \( \mathcal{L}_{st+exc} \)

Now, it comes to define a denotational semantics for the \( \text{IMP}+\text{Exc} \) language, with the combined decorated logic for the state and the exception \( \mathcal{L}_{st+exc} \) as the target language. Recall that by doing this, we aim to prove some (strong) equalities between terminating programs written in \( \text{IMP}+\text{Exc} \) with respect to the state and the exception effects.

In \( \text{IMP}+\text{Exc} \), the values that can be stored in any location (variable) \( i \) are just integers. So that any occurrence of \( (V_i) \) in term signatures of \( \mathcal{L}_{st+exc} \) is replaced by \( \mathbb{Z} \). I.e., \( \text{lookup}^{(1,0)}: I \rightarrow \mathbb{Z} \) and \( \text{update}^{(2,0)}: \mathbb{Z} \rightarrow I \). We now define a denotational semantics of \( \text{IMP}+\text{Exc} \) expressions over combined decorated settings using two translator functions \( \text{daExp} \) and \( \text{dbExp} \). The former takes an arithmetic expression as input and outputs a decorated term of type \( \text{term} \mathbb{Z} I \), while the latter takes a Boolean expression and returns a decorated term of type \( \text{term} \mathbb{B} I \):
dbExp (b₁ ∧ b₂) ⇒ (tpure andB)_{0,0} \circ \langle dbExp b₁, dbExp b₂ \rangle_{1,0}

dbExp (b₁ ∨ b₂) ⇒ (tpure orB)_{0,0} \circ \langle dbExp b₁, dbExp b₂ \rangle_{1,0}

dbExp (¬b) ⇒ (tpure notB)_{0,0} \circ dbExp b_{d,0}

In "dbExp b" (6th line above on the left), b can be either of the Boolean expressions true and false. The constructor tpure is applied to given unary and binary functions. For instance add: \((\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}\) takes an instance of an integer tuple and returns their sum. To see the definition of these functions in a Coq implementation, please check out this file \(^{(*)}\).

Remark 7.1. An expression in in IMP+Exc can have memory access right (i.e., a variable x) but can never throw or catch exceptions. To calculate the decoration d of an arithmetic expression pair, i.e., \langle daExp a₁, daExp a₂ \rangle_{1,0}, we use the following strategy:

d := \text{let } f^{d,1} = daExp(a₁) \text{ in } \text{let } g^{d,0} = daExp(a₂) \text{ in } \text{max}(d₁, d₂).

The same strategy follows for Boolean expressions, too.

We have some additional rules to make use of some pure algebraic operations in the combined decorated setting presented in Figure 18 where the pure term \text{lpbl } f : \text{I} \to \text{I}, within the rule (imp-li), is used to bridge successive loop iterations as long as the loop conditional evaluates into decorated logic’s true (constant true). Also, the pure term \text{pbl } b : \text{B} \to \text{I+I} forms a bridge between the usual Boolean data type and its correspondence in the decorated settings which is the type \text{I+I}.

\begin{align*}
\text{lpbl } (b : \text{term } \text{I} (\text{I+I})) (f : \text{term } \text{I}) & := \text{tpure } (\lambda x : \text{I}. x). \\
\text{pbl } & := \text{tpure } (\text{bool to two}). \\
\text{where } \text{bool to two } (b : \text{bool}) & := \text{if } b \text{ then } (\text{inl void}) \text{ else } (\text{inr void}).
\end{align*}

such that void : \text{I} is the unique constructor of the type \text{I}, and

\text{inl, inr : I} \to (\text{I+I}) \text{ are the canonical inclusions.}

The rule (imp₈) (functional extensionality), in Figure 18, is to say that if two pure functions on Coq side are point-wise equal, then they are strongly equal in the decorated setting. Here we take them strongly equal since strong and weak equalities are indistinguishable when the related terms are pure. The idea is to be able to use Coq’s Leibniz equality as the strong equality in the decorated setting.

Note also that in (imp₃) and (imp₄) by replacing false into true we get (imp₉) and (imp₁₀) that are not explicitly stated in Figure 18.

Lemma 7.2. pbl_{0,0} \circ (constant false)_{0,0} \equiv in₂.

Proof: unfolding all term definitions, we have \text{tpure } (\lambda b : \text{bool}. \text{if } b \text{ then } (\text{inl void}) \text{ else } (\text{inr void})) \circ \text{tpure } (\lambda _: \text{void.true}) \equiv \text{tpure inl}. Now, we obtain \forall x : \text{I}, \text{inl void} = \text{inl x} by first rewriting \text{tcomp} from left to right, and then applying \text{imp₈} which is trivial since Leibniz equality ‘=’ is reflexive.

Lemma 7.3. pbl_{0,0} \circ (constant true)_{0,0} \equiv in₁.

\(^{(*)}\) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/Functions.v
IMP with exceptions over decorated logic

\[(\text{imp}_1)\]
\[
\forall p, q : \mathbb{Z}, \ (f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z})
\]
\[\text{tpure} \circ (\text{constant}\ p, \text{constant}\ q)_1 \equiv (\text{constant}\ f(p,q))\]

\[(\text{imp}_2)\]
\[
\forall p, q : \mathbb{Z}, \ (f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}) \ f(p,q) = \text{false}
\]
\[\text{tpure} \circ (\text{constant}\ p, \text{constant}\ q)_1 \equiv \text{constant}\ \text{false}\]

\[(\text{imp}_4)\]
\[
\forall p, q : \mathbb{B}, \ (f : \mathbb{B} \times \mathbb{B} \to \mathbb{B}) \ f(p,q) = \text{false}
\]
\[\text{tpure} \circ (\text{constant}\ p, \text{constant}\ q)_1 \equiv \text{constant}\ \text{false}\]

\[(\text{imp}_6)\]
\[
(\forall x, \ f x = g x) \quad \text{tpure}\ f \equiv \text{tpure}\ g
\]

**Proof:** It follows the same steps with the proof of Lemma 7.2 \(\Box\)

**Remark 7.4.** See this file \(^{(xi)}\) for the Coq certified proofs of the Lemmas 7.2 and 7.3.

The fact that IMP+Exc commands are of type \(\mathbb{I} \to \mathbb{I}\), in \(\text{throw}^{(0,1)} : [a]^{(0,0)} \circ \text{tag}^{(0,1)} : \text{EV}_\varepsilon \to \mathbb{Y}\), we replace \(\text{EV}_\varepsilon\) and \(\mathbb{Y}\) with \(\mathbb{I}\). This means that we stick to a single exceptional value (parameter), for each exception name \(\varepsilon \in \text{ENAME}\).

Below, we recursively define the IMP+Exc commands within \(\mathcal{L}_{\text{int}+\text{exc}}\) using a translator function \(d\text{Cmd}\) which establishes a decorated term of type \(\text{term}\ \mathbb{I} \to \mathbb{I}\) out of an input command:

\[
\begin{align*}
\text{dCmd}\ (\text{SKIP}) & \Rightarrow (\text{id}\ \mathbb{I})^{(0,0)} \\
\text{dCmd}\ (x \triangleq a) & \Rightarrow (\text{update}_x)^{(2,0)} \circ (\text{daExp}\ a)^{(d_1,0)} \\
\text{dCmd}\ (c_1;c_2) & \Rightarrow (d\text{Cmd}\ c_2)^{(d_1,d_2)} \circ (d\text{Cmd}\ c_1)^{(k_1,k_2)} \\
\text{dCmd}(\text{if}\ b\ \text{then}\ c_1\ \text{else}\ c_2) & \Rightarrow \left[\text{dCmd}\ c_1\ |\ \text{dCmd}\ c_2\right]_1^{(d_1,d_2)} \circ (\text{pbl})^{(0,0)} \circ (\text{dbExp}\ b)^{(d_3,0)} \\
\text{dCmd}(\text{while}\ b\ \text{do}\ c) & \Rightarrow \left[l\text{pi}\ (\text{pbl} \circ (\text{dbExp}\ b))\ (\text{dCmd}\ c)\right]_1^{(d_1,d_2)} \circ (\text{pbl})^{(0,0)} \circ (\text{dbExp}\ b)^{(d_3,0)} \\
\text{dCmd}(\text{THROW}\ e) & \Rightarrow \text{throw}\ e^{(0,1)} \\
\text{dCmd}(\text{TRY}\ c_1\ \text{CATCH}\ e\ \Rightarrow\ c_2) & \Rightarrow \text{try}\ (\text{dCmd}\ c_1)\ \text{catch}\ (e\ \Rightarrow\ (\text{dCmd}\ c_2))^{(d_1,d_2)}
\end{align*}
\]

**Remark 7.5.** To calculate the decorations \(d_1\) and \(d_2\) (or \(k_1\) and \(k_2\)), we use the following strategy:

\[
\begin{align*}
d_1 & := \text{let}\ f^{(d'_1,d'_2)} = d\text{Cmd}(c_1)\ \text{in}\ \text{let}\ g^{(d'_3,d'_4)} = d\text{Cmd}(c_2)\ \text{in}\ \max(d'_1,d'_2). \\
d_2 & := \text{let}\ f^{(d'_1,d'_2)} = d\text{Cmd}(c_1)\ \text{in}\ \text{let}\ g^{(d'_3,d'_4)} = d\text{Cmd}(c_2)\ \text{in}\ \max(d'_3,d'_4).
\end{align*}
\]

\(^{(xi)}\) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/Derived_co_Pairs.v#L122-L133
For the strategy to calculate $d_3$, see Remark 7.1. Also, recall Definition 5.2: translation of any IMP+Exc command cannot be a public catcher, even the one for TRY/CATCH. Thus, $d_{\text{cmd}}$ function outputs terms at most with decoration (1) with respect to the exception effect.

In Figure 19, the diagram on the left schematizes the command $\text{if } b \text{ then } c_1 \text{ else } c_2$: if the Boolean expression $dbExp \ b$ evaluates into (constant $true$) then by Lemma 7.3, we have the command $c_1$ in execution, $c_2$ otherwise by Lemma 7.2. As for the loops, it is well known that as long as the looping condition evaluates into (constant $true$), loop body gets executed. This is depicted in Figure 19 (the diagram on the right), as the arrow $\text{lpi } b \ c$ is each time replaced by the whole diagram itself. The rule (imp-li) allows us to do so. If the looping condition evaluates into (constant $false$), using Lemma 7.2, we then have the term $id_1$ in execution forcing the loop to terminate. Recall that the case distinction in the diagrams are provided by the term inclusions.

Figure 20 respectively visualizes the formal behaviors of $\text{THROW}$ and TRY/CATCH commands where the basis is the core decorated terms for the exception effect. They are formulated as in Definitions 5.1 and 5.2 with a single difference in their signatures: domains and co-domains are set to $\mathbb{1}$.

We now encode the IMP+Exc denotational semantics, with the $\mathcal{L}_{\text{st-exc}}$ as the target language, in Coq. Arithmetic and Boolean expressions are inductively forming new Coq Types, called $\text{aExp}$ and $\text{bExp}$ respectively. As for the type constructors, we use the syntactic operators given as parts of $aexp$ and $bexp$ in Figure 2. The difference lies in the naming: notations are translated into plain text. It is easy to match them one another as they are given in the same order. Notice also that the implementation of the constant Boolean expressions $true$ and $false$ are subsumed by the constructor $b\text{const}$.
**IMP with exceptions over decorated logic**

Inductive `aExp` : Type ≜
| `aconst` : Z → aExp |
| `var` : Loc → aExp |
| `plus` : aExp → aExp → aExp |
| `subtr` : aExp → aExp → aExp |
| `mult` : aExp → aExp → aExp.

Inductive `bExp` : Type ≜
| `bconst` : bool → bExp |
| `eq` : aExp → aExp → bExp |
| `neq` : aExp → aExp → bExp |
| `gt` : aExp → aExp → bExp |
| `lt` : aExp → aExp → bExp |
| `ge` : aExp → aExp → bExp |
| `le` : aExp → aExp → bExp |
| `and` : bExp → bExp → bExp |
| `or` : bExp → bExp → bExp |
| `neg` : bExp → bExp.

We interpret the functions `daExp` and `dbExp` in Coq using following fixpoints:

Fixpoint `daExp` (e: aExp) : term Z unit ≜
match e with
| `aconst` n ⇒ constant n |
| `var` x ⇒ lookup x |
| `plus` a1 a2 ⇒ tpure add o pair (daExp a1) (daExp a2) |
| `subtr` a1 a2 ⇒ tpure subt o pair (daExp a1) (daExp a2) |
| `mult` a1 a2 ⇒ tpure mlt o pair (daExp a1) (daExp a2) |
end.

Fixpoint `dbExp` (e: bExp) : term bool unit ≜
match e with
| `bconst` n ⇒ constant n |
| `eq` a1 a2 ⇒ tpure chkeq o pair (daExp a1) (daExp a2) |
| `neq` a1 a2 ⇒ tpure chkneq o pair (daExp a1) (daExp a2) |
| `gt` a1 a2 ⇒ tpure chkgt o pair (daExp a1) (daExp a2) |
| `lt` a1 a2 ⇒ tpure chklt o pair (daExp a1) (daExp a2) |
| `ge` a1 a2 ⇒ tpure chkeq o pair (daExp a1) (daExp a2) |
| `le` a1 a2 ⇒ tpure chkle o pair (daExp a1) (daExp a2) |
| `and` b1 b2 ⇒ tpure andB o pair (dbExp b1) (dbExp b2) |
| `or` b1 b2 ⇒ tpure orB o pair (dbExp b1) (dbExp b2) |
| `neg` b ⇒ tpure notB o (dbExp b) |
end.

We follow a similar idea to implement commands. We inductively define a Coq type `Cmd` of IMP+Exc commands whose constructors are the members of IMP+Exc command set as presented in Figures 2 and 6. Notice that some commands are encoded with different names. I.e., the assignment command ‘≜’ is called `assign`, the sequencing command ‘;’ is called `sequence` while “if then else” block is named `cond` in the implementation. It is easy to match them one another since they are presented in the same order.

Inductive `Cmd` : Type ≜
| `skip` : Cmd |
| `sequence` : Cmd → Cmd → Cmd |
| `assign` : Loc → aExp → Cmd |
| `cond` : bExp → Cmd → Cmd → Cmd |
| `while` : bExp → Cmd → Cmd |
| `TRY_CATCH` : EName → Cmd → Cmd → Cmd. |
We now interpret the \( \text{dCmd} \) function in Coq using the below fixpoint:

\[
\text{Fixpoint } \text{dCmd} \ (c \colon \text{Cmd}) : (\text{term} \ \text{unit} \ \text{unit}) \triangleq
\begin{align*}
\text{match } c \text{ with} \\
| \text{skip} & \Rightarrow (@\text{id} \ \text{unit}) \\
| \text{sequence } c_0 \ c_1 & \Rightarrow (\text{dCmd } c_1) \circ (\text{dCmd } c_0) \\
| \text{assign } j \ e_0 & \Rightarrow (\text{update } j) \circ (\text{daExp } e_0) \\
| \text{cond } b \ c_2 \ c_3 & \Rightarrow \text{copair} (\text{dCmd } c_2) \circ (\text{dCmd } c_3) \circ (\text{pbl} \circ (\text{dbExp } b)) \\
| \text{while } b \ c_4 & \Rightarrow (\text{copair} (\text{lpj} (\text{pbl} \circ (\text{dbExp } b)) \circ (\text{dCmd } c_4)) \circ (\text{dCmd } c_4)) \circ (\text{id} \ \text{unit})) \circ (\text{pbl} \circ (\text{dbExp } b)) \\
| \text{THROW } e & \Rightarrow (\text{throw } \text{unit} \ e) \\
| \text{TRY_CATCH } e \ c_1 \ c_2 & \Rightarrow (\text{try_catch } e \ (\text{dCmd } c_1) \ (\text{dCmd } c_2))
\end{align*}
\]

Now, we retain sufficient material to state and prove equivalences between programs written in \( \text{IMP+Exc} \). Also, the discussed Coq implementation allows us to certify them in Coq.

### 7.1 Program equivalence proofs

In this section, we finally prove equivalences of several programs written in \( \text{IMP+Exc} \), using the denotational semantics characterized within the scope of the logic \( \mathcal{L}^{\text{st+exc}} \). Note that for the sake of simplicity, we will use \( u_x \), \( l_x \), \( (t \ \text{op}) \) and \( (c \ p) \), respectively.

#### Remark 7.6

Recall that the use of term products is to impose some order of term evaluation on the mutable state. \( \text{IMP+Exc} \) specific properties of the mutable state are slightly different than their generic versions (mentioned in Section 4.1) due to the fact that the language does not allow parallel term evaluations, meaning that every term is evaluated in the given sequence. Therefore, we no more need to use term products in property statements. The properties we use throughout the following proofs are re-stated in Figure 21. The full certified Coq proofs of these properties can be found here \( ^{xii} \).

1. interaction update-update \( \forall x \in \text{Loc}, \ p, q \colon \mathbb{Z}, \ u_x \circ (c \ p) \circ u_x \circ (c \ q) \equiv u_x \circ (c \ p) \)
2. commutation update-update \( \forall x \neq y \in \text{Loc}, \ p, q \colon \mathbb{Z}, \ u_x \circ (c \ p) \circ u_y \circ (c \ q) \equiv u_y \circ (c \ q) \circ u_x \circ (c \ p) \)
3. commutation-lookup-constant-update \( \forall x \in \text{Loc}, \ p, q \in \mathbb{Z}, \ (l_x \ (c \ q))_1 \circ u_x \circ (c \ p) \equiv (c \ p) \circ (c \ q)_1 \circ u_x \circ (c \ p) \)

**Figure 21:** Primitive properties of the state: \( \text{IMP+Exc} \) specific

#### Remark 7.7

Below, we state three lemmata using the \( \text{IMP+Exc} \) notation introduced in Figures 2 and 6. However, we introduce a new set of notations for the Coq encoding to increase the readability score: browse this set of notations here \( ^{xiii} \) where, i.e., the assign command is denoted by ‘::=’ while the sequence command by ‘;;’. These notations do not appear throughout the paper, but might be of help in reading the lemma statements in the Coq encoding. Notice also that they are not so pretty, due to the fact that Coq internally reserves prettier notations for other issues.

Another point here to notice is that the proofs in the following might be long and hard to follow. If you find it so, please try reading the Coq codes. They are written in parallel with the ones on the paper. Starting from Lemma 7.10, we give overall explanations about the way we compute the proof using our semantics

\( ^{xii} \) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/Proofs.v  
\( ^{xiii} \) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/IMPEX_to_COQ.v#L185-L205
before diving it into the detailed rule applications. Note also that proofs are chosen to be presented in a way that the sides of the equations are simplified until obtaining a trivial equation to solve.

**Remark 7.8.** All the statements we prove below are strong equations. The reason is that IMP+Exc (or IMP) language does not have a `return` command. Thus, one cannot compare the values that two programs return. When we use some combined decorated logic as a target language for the semantics of another language with the `return` command (i.e., the C language), then it would make sense to prove sentences with weak equations. Recall also that any strong equation can be seen as a weak equation.

**Lemma 7.9.** For all exceptionally pure commands \( f, g \) (\( \text{doesNotThrowTC}(f) = \text{true} \), \( \text{doesNotThrowTC}(g) = \text{true} \) and \( b \in \{\text{true}, \text{false}\} \), if program pieces \( \text{prog1} \) and \( \text{prog2} \) are given as in the following listings, then \( \text{dCmd}(\text{prog1}) \equiv \equiv \text{dCmd}(\text{prog2}) \).

**Listing 1: prog1**

```plaintext
/* prog1 */
if b then f else g;
```

**Listing 2: prog2**

```plaintext
/* prog2 */
if b then (if b then f else g) else g;
```

Note that the function \( \text{doesNotThrowTC} : \text{cmd} \rightarrow \text{Bool} \) takes any command, recursively checks whether the input involves either \( \text{THROW} \) or \( \text{TRY/CATCH} \), and returns \( \text{true} \) if that is the case; \( \text{false} \) otherwise. Browse this function, in a Coq implementation, here \(^{(14V)}\).

**Proof:** We sketch the diagrams of both programs below:

![Diagram](image)

where \( k = (\text{if } b \text{ then } f \text{ else } g) \). The statement we would like to prove is

\[
[f|g]_1 \circ \text{pbl} \circ c \ b \equiv \equiv [k|g]_1 \circ \text{pbl} \circ c \ b. \tag{1}
\]

Using the rules of the logic \( L_{st+exc} \), in the below given order, the idea is to simplify both sides of the statement into the same shape with respect to the equality sort \( \equiv \equiv \). The proof proceeds by a case analysis on \( b \).

If \( b = \text{false} \), by unfolding the definitions of \( \text{pbl} \) and \( c \ f \ a l s e \), we have

\[
[f|g]_1 \circ t (\text{bool_to_two}) \circ t (\lambda x : \text{unit}. \text{false}) \equiv \equiv [k|g]_1 \circ t (\text{bool_to_two}) \circ t (\lambda x : \text{unit}. \text{false}). \tag{2}
\]

We rewrite \( t \circ \text{comp} \) on both sides, and get

\[
[f|g]_1 \circ t (\lambda x : \text{unit.bool_to_two} \ \text{false}) \equiv \equiv [k|g]_1 \circ t (\lambda x : \text{unit.bool_to_two} \ \text{false}). \tag{3}
\]

\(^{(14V)}\) \url{https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/IMPEX_to_COQ.v#L148}
Now, we cut
\[ t(\lambda x:\text{unit.bool\_to\_two\_false}) \equiv in_2 \]  
\hspace{1cm} (4)
and rewrite it back in the goal. So that we obtain
\[ f \circ in_2 \equiv g \circ in_2. \]  
\hspace{1cm} (5)
Then, we use (s_lcopair_eq), and finally have \( g \equiv g \) which is trivial since \( \equiv \) is reflexive. It remains to show that the cut statement in Equation 4 holds. By simplifying \( t(\lambda x:\text{unit.bool\_to\_two\_false}) \) and unfolding \( in_2 \), we have
\[ t(\lambda x:\text{unit.inr\_x}) \equiv t(\text{inr}). \]  
\hspace{1cm} (6)
Now, we apply (imp_b) and get
\[ \forall x:\text{unit}, \text{inr\_x} = \text{inr\_x} \]  
\hspace{1cm} (7)
which is trivial since the Leibniz equality ‘=’ is reflexive.

If \( b = \text{true} \), by following above procedure with \( \text{true} \) (instead of \( \text{false} \)) we first handle
\[ f \circ in_1 \equiv g \circ in_1, \]  
\hspace{1cm} (8)
and then freely convert \( \equiv \) into \( \equiv \sim \). There, rewriting the rule (w_lcopair_eq) yields \( f \equiv \sim \text{k} \). We unfold \( k \) with \( b = \text{true} \) and get
\[ f \equiv \sim f \circ in_1. \]  
\hspace{1cm} (9)
Now by rewriting (w_lcopair_eq), we have \( f \equiv \sim f \). This is again trivial, since the equality sort \( \equiv \sim \) is reflexive.

**Lemma 7.10.** For all \( x : \text{Loc} \), if program pieces \( \text{prog}_3 \) and \( \text{prog}_4 \) are given as in the following listings, then \( d\text{Cmd} (\text{prog}_3) \equiv d\text{Cmd} (\text{prog}_4) \).

**Proof:** In the proof structure we intend to reduce \( \text{prog}_3 \), first dealing with the pre-loop assignments and the looping pre-condition. Since it evaluates into \( \text{true} \), 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 \( \text{false} \). Finally, we explain the program termination and show that \( \text{prog}_3 \) does exactly the same state manipulation with \( \text{prog}_4 \). Note also that we do not need to check the results they returned, since all \( \text{IMP+Exc} \) commands, thus programs, return \( \text{void: U} \).
Below is the sketch of prog3:

Where $f = (x \equiv x + 4)$ and $b = (x < 11)$. Using the rules of the logic $\mathcal{L}_{st+exc}$, we simplify this diagram into the one given below with respect to the equality sort $\equiv:\equiv$:

$$\begin{array}{c}
\triangleright \\
\end{array}$$

which is actually prog4 when sketched.

1. Initially, we have

$$[(lpi \; b \; f) \circ f | id_1] \circ pbl \circ (t <) \circ (t, (c \; 11)) \circ u_x \circ (c \; 2) \equiv u_x \circ (c \; 14). \quad (10)$$

Let us simplify it as far as possible. By rewriting commutation – lookup – constant – update (see Figure 21), we obtain

$$[(lpi \; b \; f) \circ f | id_1] \circ pbl \circ (t <) \circ (c \; 2, (c \; 11)) \circ u_x \circ (c \; 2) \equiv u_x \circ (c \; 14). \quad (11)$$

Since the looping pre-condition $(t <) \circ (c \; 2, (c \; 11))$ evaluates into $(c \; true)$, and due to (imp3), we have

$$[(lpi \; b \; f) \circ f | id_1] \circ pbl \circ (c \; true) \circ u_x \circ (c \; 2) \equiv u_x \circ (c \; 14). \quad (12)$$

By rewriting the Lemma 7.3, we get

$$[(lpi \; b \; f) \circ f | id_1] \circ in_1 \circ u_x \circ (c \; 2) \equiv u_x \circ (c \; 14). \quad (13)$$

Here, we first convert $\equiv \equiv$ into $\equiv \sim$ then rewrite (w_lcopair_eq), and end up with

$$(lpi \; b \; f) \circ f \circ u_x \circ (c \; 2) \equiv u_x \circ (c \; 14) \quad (14)$$

in which the second appearance of $f$ unfolds into

$$(lpi \; b \; f) \circ u_x \circ (t +) \circ (t, c \; 4) \circ u_x \circ (c \; 2) \equiv u_x \circ (c \; 14). \quad (15)$$
Since, there is no exceptional case, we are freely back to \( \equiv \). By rewriting commutation − lookup − constant − update, we obtain

\[
(lpi b f) \circ u_x \circ (t +) \circ \langle c 2, c 4 \rangle \circ u_x \circ (c 2) \equiv u_x \circ (c 14).
\] (16)

The rule \((\text{imp}_1)\) gives

\[
(lpi b f) \circ u_x \circ (c 6) \circ u_x \circ (c 2) \equiv u_x \circ (c 14).
\] (17)

Now, we rewrite the lemma interaction−update−update (see Figure 21) and get

\[
(lpi b f) \circ u_x \circ (c 6) \equiv u_x \circ (c 14).
\] (18)

2. For the second loop iteration, rewriting \((\text{imp}-\text{li})\) gives

\[
[(lpi b f) \circ f[\text{id}_1] \circ \text{pbl} \circ (t <) \circ \langle l_x, (c 11) \rangle \circ u_x \circ (c 6) \equiv u_x \circ (c 14)].
\] (19)

where looping pre-condition evaluates into \((c \text{ true})\). Therefore, we iterate the above procedure, given in the step 1, once again and derive

\[
(lpi b f) \circ u_x \circ (c 10) \equiv u_x \circ (c 14).
\] (20)

3. In the third iteration, rewriting the \((\text{imp}-\text{li})\) gives

\[
[(lpi b f) \circ f[\text{id}_1] \circ \text{pbl} \circ (t <) \circ \langle l_x, (c 11) \rangle \circ u_x \circ (c 10) \equiv u_x \circ (c 14)].
\] (21)

As in step 2, the looping pre-condition evaluates into \((c \text{ true})\) forcing us to reiterate the above procedure, given in the step 1, which results in

\[
(lpi b f) \circ u_x \circ (c 14) \equiv u_x \circ (c 14).
\] (22)

4. In the fourth step, rewriting the \((\text{imp}-\text{li})\) gives

\[
[(lpi b f) \circ f[\text{id}_1] \circ \text{pbl} \circ (t <) \circ \langle l_x, (c 11) \rangle \circ u_x \circ (c 14) \equiv u_x \circ (c 14)].
\] (23)

By rewriting commutation − lookup − constant − update, we obtain

\[
[(lpi b f) \circ f[\text{id}_1] \circ \text{pbl} \circ (t <) \circ \langle (c 14), (c 11) \rangle \circ u_x \circ (c 14) \equiv u_x \circ (c 14)].
\] (24)

Finally here, the looping pre-condition \((t <) \circ \langle (c 14), (c 11) \rangle\) evaluates into \((c \text{ false})\) yielding

\[
[(lpi b f) \circ f[\text{id}_1] \circ \text{pbl} \circ (c \text{ false}) \circ u_x \circ (c 14) \equiv u_x \circ (c 14)].
\] (25)

We rewrite the Lemma 7.2, and get

\[
[(lpi b f) \circ f[\text{id}_1] \circ \text{in}_2 \circ u_x \circ (c 14) \equiv u_x \circ (c 14)].
\] (26)

Now we rewrite \((s_{\text{lcopair_eq}})\) and handle

\[
id_1 \circ u_x \circ (c 14) \equiv u_x \circ (c 14)
\] (27)

which is trivial, since the identity term disappears when to compose and the equality sort \( \equiv \) is reflexive.
Lemma 7.11. For each \(x, y : \text{Loc}, e : \text{EName}\), if program pieces \(\text{prog5}\) and \(\text{prog6}\) are given as in the following listings, then \(\text{dCmd}(\text{prog5}) \equiv \text{dCmd}(\text{prog6})\).

Listing 5: \(\text{prog5}\)

```plaintext
/* prog5 */
x ≜ 1;
y ≜ 20;
TRY(
    while (true)
    do
        ( if (x ? <= 0) then (THROW e)
        else x ≜ x - 1
    )
) CATCH e ⇒ (y ≜ 7);
```

Listing 6: \(\text{prog6}\)

```plaintext
/* prog6 */
x ≜ 0;
y ≜ 7;
```

Proof: In the 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 which is followed by the abrupt loop termination. Finally, in the fourth step, we explain the exception recovery and the program termination. Below is the sketch of \(\text{prog5}\):

\[
\text{Diagram of prog5}
\]

where \(b = (x \leq 0), c_0 = (x \triangleq 1; y \triangleq 20), c_1 = (if (x \leq 0) then (THROW e) else (x \triangleq x - 1)), c_2 = (x \triangleq x - 1), c_3 = (y \triangleq 7)\). The dotted arrows depict the normal loop iterations while dashed ones are to identify the program behavior after the exception of name \(e\) is raised. Using the rules of the logic \(\mathcal{L}_{\text{St+Exc}}\), we can reduce the above diagram into the one given below with respect to the equality sort \(\equiv\):

\[
\text{Diagram of reduced prog5}
\]
which is actually the prog6 when sketched.

1. Initially, we have

\[
\downarrow \left( [id_1]_{c_3 \circ \text{untag}_e} \circ \text{in}_1 \circ \left[ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right] \circ \text{pbl} \circ b \circ id_1 \right) \circ \text{pbl} \circ (c \text{ true})
\]

\[
\circ \text{u}_x \circ (c \ 20) \circ \text{u}_x \circ (c \ 1) \equiv \equiv \text{u}_y \circ (c \ 7) \circ \text{u}_x \circ (c \ 0) . \quad (28)
\]

We convert \(\equiv\) into \(\equiv\), then rewrite the (w_downcast) rule and get

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ \left[ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right] \circ \text{pbl} \circ b \circ id_1 \right]
\]

\[
\circ \text{pbl} \circ (c \text{ true}) \circ \text{u}_x \circ (c \ 20) \circ \text{u}_x \circ (c \ 1) \equiv \equiv \text{u}_y \circ (c \ 7) \circ \text{u}_x \circ (c \ 0) . \quad (29)
\]

Rewriting commutation-update-update, on both sides, gives

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ \left[ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right] \circ \text{pbl} \circ b \circ id_1 \right]
\]

\[
\circ \text{pbl} \circ (c \text{ true}) \circ \text{u}_x \circ (c \ 1) \circ \text{u}_y \circ (c \ 20) \equiv \equiv \text{u}_x \circ (c \ 0) \circ \text{u}_y \circ (c \ 7) . \quad (30)
\]

Rewriting Lemma 7.3 yields

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ \left[ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right] \circ \text{pbl} \circ b \circ id_1 \right]
\]

\[
\circ \text{in}_1 \circ \text{u}_x \circ (c \ 1) \circ \text{u}_y \circ (c \ 20) \equiv \equiv \text{u}_x \circ (c \ 0) \circ \text{u}_y \circ (c \ 7) . \quad (31)
\]

2. Now we rewrite the rule (w_lcopair_eq), and handle

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right]
\]

\[
\circ \text{pbl} \circ b \circ \text{u}_x \circ (c \ 1) \circ \text{u}_y \circ (c \ 20) \equiv \equiv \text{u}_x \circ (c \ 0) \circ \text{u}_y \circ (c \ 7) . \quad (32)
\]

By unfolding b, we have

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right]
\]

\[
\circ \text{pbl} \circ (t \ \leq) \circ \langle \text{true} \circ (c \ 0) \rangle \circ \text{u}_x \circ (c \ 1) \circ \text{u}_y \circ (c \ 20) \equiv \equiv \text{u}_x \circ (c \ 0) \circ \text{u}_y \circ (c \ 7) . \quad (33)
\]

By rewriting the lemma commutation-lookup-constant-update, we obtain

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right]
\]

\[
\circ \text{pbl} \circ (t \ \leq) \circ \langle (c \ 1) , (c \ 0) \rangle \circ \text{u}_x \circ (c \ 1) \circ \text{u}_y \circ (c \ 20) \equiv \equiv \text{u}_x \circ (c \ 0) \circ \text{u}_y \circ (c \ 7) . \quad (34)
\]

We rewrite the rule (imp2), and get

\[
\left[ id_1 \circ [c_3 \circ \text{untag}_e] \circ \text{in}_1 \circ (lpi (c \text{ true}) \circ c_1) \circ [\text{tag}_e]_{c_2} \right] \circ \text{pbl} \circ (c \text{ false}) \circ \text{u}_x \circ (c \ 1) \circ \text{u}_y \circ (c \ 20) \equiv \equiv \text{u}_x \circ (c \ 0) \circ \text{u}_y \circ (c \ 7) . \quad (35)
\]
Rewriting the Lemma 7.2 yields
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \\
& \circ [\mid]_1 \circ \text{tag}_{\alpha} \mid c_2 \circ \text{in}_2 \circ u_x \circ (c 1) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (36)
\end{align*}
\]

We now rewrite (s_icopair_eq) which gives
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \\
& \circ c_2 \circ u_x \circ (c 1) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (37)
\end{align*}
\]

Here, by unfolding c_2, we have
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \circ u_x \circ (t -) \circ (1_x, (c 1)) \\
& \circ u_x \circ (c 1) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (38)
\end{align*}
\]

Rewriting the lemma commutation - lookup - constant - update gives
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \circ u_x \circ (t -) \circ ((c 1), (c 1)) \\
& \circ u_x \circ (c 1) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (39)
\end{align*}
\]

We rewrite (imp_1), and get
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \\
& \circ u_x \circ (c 0) \circ u_x \circ (c 1) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (40)
\end{align*}
\]

We again rewrite the lemma commutation-update-update, and obtain
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \circ u_x \circ (c 0) \\
& \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (41)
\end{align*}
\]

3. We re-iterate the loop via (imp-li), and have
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ [(\text{lpi} (c \text{ true}) c_1) \circ \text{id}] \\
& \circ \text{pbl} \circ (c \text{ true}) \circ u_x \circ (c 0) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 7). \quad (42)
\end{align*}
\]

We rewrite Lemma 7.3, (w_icopair_eq), then unfold c_1, and get:
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \circ \text{throw e I} \circ c_2 \\
& \circ \text{pbl} \circ (t \leq) \circ (1_x, (c 0)) \circ u_x \circ (c 0) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 20). \quad (43)
\end{align*}
\]

By rewriting commutation - lookup - constant - update, (imp_1) and Lemma 7.3, we have
\[
\begin{align*}
\text{id}_1 & \mid c_3 \circ \text{untag}_{\alpha} \circ \text{in}_1 \circ (\text{lpi} (c \text{ true}) c_1) \circ \text{throw e I} \circ c_2 \circ \text{in}_1 \\
& \circ u_x \circ (c 0) \circ u_y \circ (c 20) \equiv \sim u_x \circ (c 0) \circ u_y \circ (c 20). \quad (44)
\end{align*}
\]
By \((w_{lcopair\_eq})\), the exception is raised:

\[
\begin{align*}
\text{id} \circ & \text{c}_3 \circ \text{untag}_e \circ \text{in}_1 \circ (\text{lpi}(\text{c truth}) \text{c}_1) \circ \text{throw}_e \text{I} \\
& \circ \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20). \quad (45)
\end{align*}
\]

Due to the raised exception, the infinite loop gets abruptly terminated at this step. Here we unfold the definition of \(\text{THROW}\) then rewrite \(\text{propagator\_propagates}\) (see Section 6.1), and get

\[
\begin{align*}
\text{id} \circ & \text{c}_3 \circ \text{untag}_e \circ \text{in}_1 \circ [1] \circ \text{tag}_e \circ \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20). \quad (46)
\end{align*}
\]

4. Here, we first cut \(\text{in}_1 \circ [1] \equiv \text{in}_2\), and rewrite it back in the equation. Thus, we have

\[
\begin{align*}
\text{id} \circ & \text{c}_3 \circ \text{untag}_e \circ \text{in}_2 \circ \text{tag}_e \circ \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7). \quad (47)
\end{align*}
\]

By rewriting \((s_{lcopair\_eq})\), we obtain

\[
\text{c}_3 \circ \text{untag}_e \circ \text{tag}_e \circ \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7). \quad (48)
\]

Since \(\text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20)\) is pure with respect to the exception, we rewrite \((eax_1)\), and get

\[
\text{c}_3 \circ \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7). \quad (49)
\]

Unfolding the definition of the command \(\text{c}_3 = (\text{u}_y \circ (\text{c} \ 7))\), we have

\[
\text{u}_y \circ (\text{c} \ 7) \circ \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7). \quad (50)
\]

We now rewrite \(\text{commutation\_update\_update}\) on the left, and handle

\[
\text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7) \circ \text{u}_y \circ (\text{c} \ 20) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7). \quad (51)
\]

Finally, it suffices to rewrite \(\text{interaction\_update\_update}\),

\[
\text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7) \equiv \text{u}_x \circ (\text{c} \ 0) \circ \text{u}_y \circ (\text{c} \ 7). \quad (52)
\]

which is trivial since the equality symbol \(\equiv\) is reflexive. However, it still remains to prove the previous cut \(\text{in}_1 \circ [1] \equiv \text{in}_2\); since everything is pure with respect to the exception, we have

\[
\text{in}_1 \circ [1] \equiv \text{in}_2. \quad (53)
\]

Now, rewriting the rule \((w_{empty})\) gives \([1]_{i+1} \equiv [1]_{i+1}\). This is trivial since the equality sort \(\equiv\) is reflexive.

The full Coq proofs of above lemmata can be found here\(^{(xv)}\), and the entire implementation there\(^{(xvi)}\).
7.2 Automating decorated proofs

A rule in a decorated logic only applies if the given term gets decorated as expected by the rule. Therefore, decoration checks are pretty important and occur pretty often. To automatize this checks, at the Coq level, we already have tactics `decorate` and `edecorate`. See them here \(^{(xvii)}\). Also, we plan to put Czajka and Kaliszyk (2018)'s CoqHammer tool in use to try automatizing such program property proofs, done within the scope of decorated logics, implemented in Coq.

7.3 On the completeness of the logic \( \mathcal{L}_{st+exc} \)

With the logic \( \mathcal{L}_{st+exc} \), no generic program properties such as

\[
dCmd(p_1) \equiv dCmd(p_2) \implies \forall ss', \text{eval} p_1 ss' = \text{eval} p_2 ss'
\]

can be proven. Here, `eval` denotes the big-step semantics of the commands until reaching `SKIP`. Only programs that admit a particular specification can be proven to be equivalent with respect to the state and exception effects. The total correctness is based on a syntactic completeness property. In a way, it is meant to make sure that we are not using too many axioms to construct a denotational semantics for the IMP$+\text{Exc}$ language using the logic \( \mathcal{L}_{st+exc} \) as the target language. This syntactic completeness property is called relative Hilbert-Post Completeness (rHPC) and elaborately defined by Dumas et al. (2015). Briefly, given two logics \( L_0 \) and \( L \) such that \( L_0 \subseteq L \) (\( L_0 \) is a sub-logic of \( L \)) and a theory \( T \) of \( L \), \( T \) is relatively Hilbert-Post complete with respect to \( L_0 \) if (1) at least one sentence is unprovable in \( T \) (not the maximal theory ensuring consistency), and (2) every theory containing \( T \) can be generated from \( T \) and some sentences from \( L_0 \). Here, \( L_0 \) can be seen as the pure logic that governs the denotational semantics of the effect-free subset of the IMP language where \( L \) is the logic that governs the denotational semantics of the superset of the IMP language after either the state or exception effect is added.

We prove, in Theorem 6.8.5 in (Ekici (2015)), that the decorated theory of exceptions is relatively Hilbert-Post complete with respect to its pure part. However, only the core part of the decorated logic for the state effect is proven to be rHPC (see Theorem 5.4.9 in Ekici (2015)). What we mean by the "core part" is the logic with no categorical pairs. Clearly, when translated to IMP denotational semantics, it corresponds to the part that governs conditionals and loops. We can conjecture that the logic is still complete in the presence of categorical pairs. However, the proof is not yet done.

In the rHPC proof of the core part for the decorated logic for the state, we first determine the canonical forms of accessors and modifiers and then show that both such forms are equivalent to some finite set of equations in the pure sublogic of \( \mathcal{L}_{st} \) with no pairs. In the presence of categorical pairs, we so far had difficulties to come up with the canonical forms for accessors and modifiers even though it is clear that such forms exist. Once we have these forms in hand, it should also be the case that the rules governing pairs suffice to prove that such forms are equivalent to finite number of equations made of terms coming from the pure counterpart of the logic \( \mathcal{L}_{st} \). We plan to study this in the near future.

It is also proven that if two theories are rHPC with respect to a (pure) logic, then the combination of these theories remains to be rHPC. Therefore, the logic \( \mathcal{L}_{st+exc} \) without the use of pairs is rHPC.

\(^{(xvii)}\) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/Decorations.v
8 Concluding remarks

We have presented frameworks for formalizing the treatment of the state and the exception effects, first separately, and then combined, using the decorated logic. Decorations describe what computational effect evaluation of a term may involve, and form a bridge between the syntax and its interpretation in reasoning about terms by making computational effects explicit in the decorated syntax. We have designed a denotational semantics for the \( \text{IMP+Exc} \) language over the combined decorated logic \( \mathcal{L}_{\text{st+exc}} \). This way, we managed to prove some strong equalities between \( \text{IMP+Exc} \) programs. We have also encoded the combined logic in the Coq proof assistant and certified related proofs.

Acknowledgements

We wish to thank the two anonymous reviewers for their careful reading of our manuscript and for insightful comments and suggestions. We thank Jean-Guillaume Dumas and Dominique Duval for their support on all aspects of the presented logics. Many thanks also to Damien Pous for his guidance on Coq related questions. This work has been partially supported by the Austrian Science Fund (FWF) grant P26201 and the European Research Council (ERC) Grant No. 714034 SMART.

References


IMP with exceptions over decorated logic


