IMP with exceptions over decorated logic

Burak Ekici

Discrete Mathematics and Theoretical Computer Science DMTCS vol. **20**:2, 2018, #11

`IMP` *with exceptions over decorated logic*

Burak Ekici

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

received 18th Apr. 2017, revised 23rd Feb. 2018, 21st Sep. 2018, accepted 5th Oct. 2018.

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, η, μ) over a base category $\mathscr{C}$ 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 $\mathscr{C}$ 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

ISSN 1365–8050

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. Petricek 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 λ-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 guaranties 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.

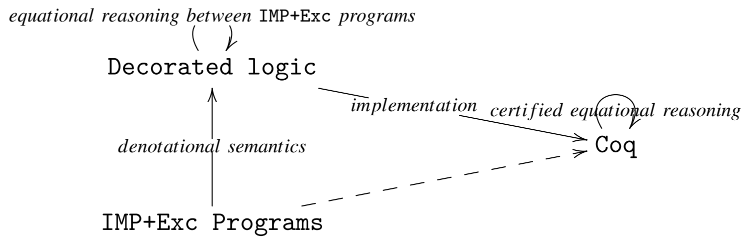


Figure 1: The approach

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 (`Z`), Boolean (`B`) and unit (`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.

$$\begin{array}{lcl} \text{aexp: } \mathtt{a}_1\ \mathtt{a}_2 & ::= & \mathtt{n} \mid \mathtt{x} \mid \mathtt{a}_1 + \mathtt{a}_2 \mid \mathtt{a}_1 - \mathtt{a}_2 \mid \mathtt{a}_1 \times \mathtt{a}_2 \\ \text{bexp: } \mathtt{b}_1\ \mathtt{b}_2 & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{a}_1 \stackrel{?}{=} \mathtt{a}_2 \mid \mathtt{a}_1 \stackrel{?}{\neq} \mathtt{a}_2 \mid \mathtt{a}_1 \stackrel{?}{>} \mathtt{a}_2 \mid \mathtt{a}_1 \stackrel{?}{<} \mathtt{a}_2 \mid \mathtt{b}_1 \wedge \mathtt{b}_2 \mid \mathtt{b}_1 \vee \mathtt{b}_2 \mid \neg \mathtt{b}_1 \\ \text{cmd: } \mathtt{c}_1\ \mathtt{c}_2 & ::= & \mathtt{SKIP} \mid \mathtt{x} \triangleq \mathtt{a}_1 \mid \mathtt{c}_1;\ \mathtt{c}_2 \mid \mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{c}_1\ \mathtt{else}\ \mathtt{c}_2 \mid \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c}_1 \end{array}$$

Figure 2: Standard `IMP` syntax

[(i)] https://github.com/ekiciburak/impex-on-decorated-logic
[(ii)] ftp://ftp-sop.inria.fr/indes/TFP15/TFP2015_submission_6.pdf

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 $\to_a : \mathtt{aexp} \times \mathtt{S} \to \mathbb{Z}$. 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 op represents the operation symbols ($+$, $-$ or $\times$) given by the standard syntax in Figure 3, while $\mathtt{op}_\mathbb{Z} : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ denotes the corresponding binary operations in $\mathbb{Z}$.

$$(\mathtt{aconst})\frac{}{(\mathtt{n},\ \mathtt{s}) \to_a \mathtt{n}} \qquad (\mathtt{var})\frac{}{(\mathtt{x},\ \mathtt{s}) \to_a \mathtt{s(x)}} \qquad (\mathtt{op-sym})\frac{(\mathtt{a_1},\ \mathtt{s}) \to_\mathtt{a} \mathtt{n_1} \qquad (\mathtt{a_2},\ \mathtt{s}) \to_\mathtt{a} \mathtt{n_2}}{(\mathtt{a_1}\ \mathtt{op}\ \mathtt{a_2}, \mathtt{s}) \to_a \mathtt{n_1}\ \mathtt{op}_\mathbb{Z}\ \mathtt{n_2}}$$

Figure 3: Big-step operational semantics for arithmetic expressions

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 $\to_b : \mathtt{bexp} \times \mathtt{S} \to \mathbb{B}$. This function simply computes a Boolean value out of an input Boolean expression and the current program state. The constant symbols true and false are Boolean operation symbols given by the standard syntax in Figure 2, while *true* and *false* are Boolean constructors. Similarly, opb represents the binary operation symbols, while $\mathtt{opb}_\mathbb{B} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ denotes the corresponding Boolean operations, and $\mathtt{neg} : \mathbb{B} \to \mathbb{B}$ is the Boolean negation.

$$(\mathtt{true})\frac{}{(\mathtt{true},\ \mathtt{s}) \to_b \mathit{true}} \qquad (\mathtt{false})\frac{}{(\mathtt{false},\ \mathtt{s}) \to_b \mathit{false}}$$

$$(\mathtt{op1})\frac{(\mathtt{b_1},\ \mathtt{s}) \to_\mathtt{b} \mathtt{v_1} \qquad (\mathtt{b_2},\ \mathtt{s}) \to_\mathtt{b} \mathtt{v_2}}{(\mathtt{b_1}\ \mathtt{opb}\ \mathtt{b_2}, \mathtt{s}) \to_b \mathtt{v_1}\ \mathtt{opb}_\mathbb{B}\ \mathtt{v_2}} \qquad (\mathtt{op2})\frac{(\mathtt{b_1},\ \mathtt{s}) \to_\mathtt{b} \mathtt{v_1}}{(\neg\ \mathtt{b_1}, \mathtt{s}) \to_b \mathtt{neg}\ \mathtt{v_1}}$$

Figure 4: Big-step operational semantics for Boolean expressions

The small-step operational semantics for evaluation of commands are given in Figure 5 where we use a small-step transition function $\rightsquigarrow : \mathtt{S} \times \mathtt{cmd} \to \mathtt{S} \times \mathtt{cmd}$ which is interpreted as *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.*

$$(\mathtt{sequence})\frac{\mathtt{s}, \mathtt{c_1} \rightsquigarrow \mathtt{s'}, \mathtt{c'_1}}{\mathtt{s}, (\mathtt{c_1}; \mathtt{c_2}) \rightsquigarrow \mathtt{s'}, (\mathtt{c'_1}; \mathtt{c_2})} \qquad (\mathtt{skip})\frac{}{\mathtt{s},\ (\mathtt{SKIP}; \mathtt{c}) \rightsquigarrow \mathtt{s}, \mathtt{c}}$$

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

$$(\mathtt{cond_1})\frac{(\mathtt{b},\ \mathtt{s}) \to_\mathtt{b} \mathit{true}}{\mathtt{s},\ (\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{c_1}\ \mathtt{else}\ \mathtt{c_2}) \rightsquigarrow \mathtt{s}, \mathtt{c_1}} \qquad (\mathtt{cond_2})\frac{(\mathtt{b},\ \mathtt{s}) \to_\mathtt{b} \mathit{false}}{\mathtt{s},\ (\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ \mathtt{c_1}\ \mathtt{else}\ \mathtt{c_2}) \rightsquigarrow \mathtt{s}, \mathtt{c_2}}$$

$$(\mathtt{while_1})\frac{(\mathtt{b},\ \mathtt{s}) \to_\mathtt{b} \mathit{true}}{\mathtt{s},\ (\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c}) \rightsquigarrow \mathtt{s}, (\mathtt{c};\ \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c})} \qquad (\mathtt{while_2})\frac{(\mathtt{b},\ \mathtt{s}) \to_\mathtt{b} \mathit{false}}{\mathtt{s},\ (\mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{c}) \rightsquigarrow \mathtt{s}, \mathtt{SKIP}}$$

Figure 5: Small-step operational semantics for commands

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

2.1 A mechanism to handle exceptions

Extending the IMP language with a mechanism that allows raising exceptions and recovering from them, we enrich the command set with THROW and 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

$$\text{cmd: } \mathtt{c_1}\ \mathtt{c_2} \quad ::= \quad \ldots \mid \mathtt{THROW}\ \mathtt{e} \mid \mathtt{TRY}\ \mathtt{c_1}\ \mathtt{CATCH}\ \mathtt{e} \Rightarrow \mathtt{c_2}$$

Figure 6: Syntax for exceptional commands

EName which exists by assumption. There is also a type $\mathtt{EV_e}$ of exceptional values (parameters) for each exception name e. The small-step operational semantics for THROW and TRY/CATCH commands are shown in Figure 7.

$$(\mathtt{throw})\frac{\mathtt{e} : \mathtt{EName}}{\mathtt{s},\ (\mathtt{THROW}\ \mathtt{e};\mathtt{c}) \rightsquigarrow \mathtt{s},\ \mathtt{THROW}\ \mathtt{e}} \quad (\mathtt{tskip})\frac{}{\mathtt{s}, \mathtt{TRY}\ \mathtt{SKIP}\ \mathtt{CATCH}\ \mathtt{e} \Rightarrow \mathtt{c} \rightsquigarrow \mathtt{s}, \mathtt{SKIP}}$$

$$(\mathtt{tstep})\frac{\mathtt{s}, \mathtt{c_1} \rightsquigarrow \mathtt{s'}, \mathtt{c'_1}}{\mathtt{s}, \mathtt{TRY}\ \mathtt{c_1}\ \mathtt{CATCH}\ \mathtt{e} \Rightarrow \mathtt{c_2} \rightsquigarrow \mathtt{s'}, \mathtt{TRY}\ \mathtt{c'_1}\ \mathtt{CATCH}\ \mathtt{e} \Rightarrow \mathtt{c_2}}$$

$$(\mathtt{tc_1})\frac{\mathtt{e} : \mathtt{EName}}{\mathtt{s}, \mathtt{TRY}\ (\mathtt{THROW}\ \mathtt{e})\ \mathtt{CATCH}\ \mathtt{e} \Rightarrow \mathtt{c} \rightsquigarrow \mathtt{s}, \mathtt{c}} \quad (\mathtt{tc_2})\frac{\mathtt{e_1}\ \mathtt{e_2} : \mathtt{EName} \quad \mathtt{e_1} \neq \mathtt{e_2}}{\mathtt{s}, \mathtt{TRY}\ (\mathtt{THROW}\ \mathtt{e_1})\ \mathtt{CATCH}\ \mathtt{e_2} \Rightarrow \mathtt{c} \rightsquigarrow \mathtt{s}, \mathtt{THROW}\ \mathtt{e_1}}$$

Figure 7: Small-step operational semantics for additional commands

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 SKIP alone indicates the termination of a program. Now, we extend this by saying THROW e is also an end but an abnormal end. Intuitively, if an exceptional value of name e is raised in the TRY block and recovered immediately in the 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. $\mathtt{e} \neq \mathtt{f}$) is being recovered in the CATCH.

In Section 7, we define denotational semantics of the 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 ($\mathscr{L}_{dec}$)

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 *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 ($\mathscr{L}_{meq}$)

The *monadic equational logic* ($\mathscr{L}_{meq}$) 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 $\mathtt{O}$ in the category interprets the type $\mathtt{X}$ in the logic, just as the usual Leibniz equality, $\mathtt{f} = \mathtt{g}$, interprets the equation $\mathtt{f} \cong \mathtt{g}$ 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 $\mathscr{L}_{meq}$. There, every term has a source and a

Grammar for the monadic equational logic:

Types:	$\mathtt{t}$	::=	$\mathtt{X} \mid \mathtt{Y} \mid \ldots$
Terms:	$\mathtt{f}, \mathtt{g}$	::=	$\mathtt{id_t} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{g} \circ \mathtt{f}$
Equations:	$\mathtt{eq}$	::=	$\mathtt{f} \cong \mathtt{g}$

Figure 8: $\mathscr{L}_{meq}$: syntax

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

congruence rules

$$(\text{refl})\frac{\mathtt{f}}{\mathtt{f} \cong \mathtt{f}} \quad (\text{sym})\frac{\mathtt{f} \cong \mathtt{g}}{\mathtt{g} \cong \mathtt{f}} \quad (\text{trans})\frac{\mathtt{f} \cong \mathtt{g} \quad \mathtt{g} \cong \mathtt{h}}{\mathtt{f} \cong \mathtt{h}} \quad (\text{replsubs})\frac{\mathtt{f_1} \cong \mathtt{f_2}\colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{g_1} \cong \mathtt{g_2}\colon \mathtt{Y} \to \mathtt{Z}}{\mathtt{g_1} \circ \mathtt{f_1} \cong \mathtt{g_2} \circ \mathtt{f_2}}$$

categorical rules

$$(\text{id})\frac{\mathtt{X}}{\mathtt{id_X}\colon \mathtt{X} \to \mathtt{X}} \quad (\text{comp})\frac{\mathtt{f}\colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}\colon \mathtt{Y} \to \mathtt{Z}}{(\mathtt{g} \circ \mathtt{f})\colon \mathtt{X} \to \mathtt{Z}} \quad (\text{ids})\frac{\mathtt{f}\colon \mathtt{X} \to \mathtt{Y}}{\mathtt{f} \circ \mathtt{id_X} \cong \mathtt{f}} \quad (\text{idt})\frac{\mathtt{f}\colon \mathtt{X} \to \mathtt{Y}}{\mathtt{id_Y} \circ \mathtt{f} \cong \mathtt{f}}$$

$$(\text{assoc})\frac{\mathtt{f}\colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}\colon \mathtt{Y} \to \mathtt{Z} \quad \mathtt{h}\colon \mathtt{Z} \to \mathtt{U}}{\mathtt{h} \circ (\mathtt{g} \circ \mathtt{f}) \cong (\mathtt{h} \circ \mathtt{g}) \circ \mathtt{f}}$$

Figure 9: $\mathscr{L}_{meq}$: rules

The *congruence rules* say that the relation '$\cong$' 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 $\mathtt{id_X}\colon \mathtt{X} \to \mathtt{X}$ for each type $\mathtt{X}$, composition is an associative operation, and composing any term $\mathtt{f}$ with $\mathtt{id}$ is $\mathtt{f}$, up to $\cong$, no matter the composition order.

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:	$\mathtt{t}$	$::=$	$\mathtt{X} \mid \mathtt{Y} \mid \dots$
Decoration for terms:	$(\mathtt{d})$	$::=$	$(0) \mid (1) \mid (2)$
Terms:	$\mathtt{f},\ \mathtt{g}$	$::=$	$\mathtt{a}^{(\mathtt{d})} \mid \mathtt{b}^{(\mathtt{d})} \mid \cdots \mid \mathtt{g} \circ \mathtt{f}^{(\mathtt{d})} \mid (\mathtt{tpure}\ \bullet)^{(0)}$
Equations:	$\mathtt{eq}$	$::=$	$\mathtt{f} \equiv \mathtt{g} \mid \mathtt{f} \sim \mathtt{g}$

Figure 10: $\mathscr{L}_{dec}$: syntax

The `tpure` is a special constructor used to introduce decorated pure terms into the logic $\mathscr{L}_{dec}$. 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 `id` is defined using the `tpure` constructor, for all types `X` as follows:

$$\mathtt{id}_{\mathtt{X}}^{(0)} \quad : \quad \mathtt{X} \to \mathtt{X} \quad := \quad \mathtt{tpure}\ (\lambda\, \mathtt{x} : \mathtt{X}.\mathtt{x} : \mathtt{X}).$$

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, through out the paper, the decorations "$\mathtt{d}_1, \mathtt{d}_2, \mathtt{d}_3, \dots$" are meant to be in the set $\{0,1,2\}$ unless otherwise stated. For instance, in the rule (wtos) below decorations $\mathtt{d}_1$ and $\mathtt{d}_2$ cannot take the value 2.

hierarchy rules

$$(\text{0-to-1})\frac{\mathtt{f}^{(0)}}{\mathtt{f}^{(1)}} \quad (\text{1-to-2})\frac{\mathtt{f}^{(1)}}{\mathtt{f}^{(2)}} \quad (\text{stow})\frac{\mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{g}^{(\mathtt{d}_2)}}{\mathtt{f}^{(\mathtt{d}_1)} \sim \mathtt{g}^{(\mathtt{d}_2)}} \quad (\text{wtos})\frac{\mathtt{f}^{(\mathtt{d}_1)} \sim \mathtt{g}^{(\mathtt{d}_2)} \quad \mathtt{d}_1, \mathtt{d}_2 \in \{0,1\}}{\mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{g}^{(\mathtt{d}_2)}}$$

congruence rules

$$(\text{refl})\frac{\mathtt{f}^{(\mathtt{d}_1)}}{\mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{f}^{(\mathtt{d}_1)}} \quad (\text{sym})\frac{\mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{g}^{(\mathtt{d}_2)}}{\mathtt{g}^{(\mathtt{d}_2)} \equiv \mathtt{f}^{(\mathtt{d}_1)}} \quad (\text{trans})\frac{\mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{g}^{(\mathtt{d}_2)} \quad \mathtt{g}^{(\mathtt{d}_2)} \equiv \mathtt{h}^{(\mathtt{d}_3)}}{\mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{h}^{(\mathtt{d}_3)}}$$

$$(\text{wrefl})\frac{\mathtt{f}^{(\mathtt{d}_1)}}{\mathtt{f}^{(\mathtt{d}_1)} \sim \mathtt{f}^{(\mathtt{d}_1)}} \quad (\text{wsym})\frac{\mathtt{f}^{(\mathtt{d}_1)} \sim \mathtt{g}^{(\mathtt{d}_2)}}{\mathtt{g}^{(\mathtt{d}_2)} \sim \mathtt{f}^{(\mathtt{d}_1)}} \quad (\text{wtrans})\frac{\mathtt{f}^{(\mathtt{d}_1)} \sim \mathtt{g}^{(\mathtt{d}_2)} \quad \mathtt{g}^{(\mathtt{d}_2)} \sim \mathtt{h}^{(\mathtt{d}_3)}}{\mathtt{f}^{(\mathtt{d}_1)} \sim \mathtt{h}^{(\mathtt{d}_3)}}$$

$$(\text{replsubs})\frac{\mathtt{f}_1^{(\mathtt{d}_1)} \equiv \mathtt{f}_2^{(\mathtt{d}_2)} : \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}_1^{(\mathtt{d}_3)} \equiv \mathtt{g}_2^{(\mathtt{d}_4)} : \mathtt{Y} \to \mathtt{Z}}{\mathtt{g}_1^{(\mathtt{d}_3)} \circ \mathtt{f}_1^{(\mathtt{d}_1)} \equiv \mathtt{g}_2^{(\mathtt{d}_4)} \circ \mathtt{f}_2^{(\mathtt{d}_2)}}$$

categorical rules

$$(\text{comp})\frac{\mathtt{f}^{(\mathtt{d}_1)} : \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}^{(\mathtt{d}_1)} : \mathtt{Y} \to \mathtt{Z}}{(\mathtt{g} \circ \mathtt{f})^{(\mathtt{d}_1)} : \mathtt{X} \to \mathtt{Z}} \quad (\text{assoc})\frac{\mathtt{f}^{(\mathtt{d}_1)} : \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}^{(\mathtt{d}_2)} : \mathtt{Y} \to \mathtt{Z} \quad \mathtt{h}^{(\mathtt{d}_3)} : \mathtt{Z} \to \mathtt{U}}{\mathtt{h}^{(\mathtt{d}_3)} \circ (\mathtt{g}^{(\mathtt{d}_2)} \circ \mathtt{f}^{(\mathtt{d}_1)}) \equiv (\mathtt{h}^{(\mathtt{d}_3)} \circ \mathtt{g}^{(\mathtt{d}_2)}) \circ \mathtt{f}^{(\mathtt{d}_1)}}$$

$$(\text{ids})\frac{\mathtt{f}^{(\mathtt{d}_1)} : \mathtt{X} \to \mathtt{Y}}{\mathtt{f}^{(\mathtt{d}_1)} \circ \mathtt{id}_{\mathtt{X}}^{(0)} \equiv \mathtt{f}^{(\mathtt{d}_1)}} \quad (\text{idt})\frac{\mathtt{f}^{(\mathtt{d}_1)} : \mathtt{X} \to \mathtt{Y}}{\mathtt{id}_{\mathtt{Y}}^{(0)} \circ \mathtt{f}^{(\mathtt{d}_1)} \equiv \mathtt{f}^{(\mathtt{d}_1)}}$$

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

Figure 11: $\mathscr{L}_{dec}$: rules

Lemma 3.2. $\forall \mathtt{f}^{(\mathtt{d}_1)} : \mathtt{X} \to \mathtt{Y}, \mathtt{g}^{(\mathtt{d}_2)} : \mathtt{Y} \to \mathtt{Z}$, *the annotation* $(\mathtt{g} \circ \mathtt{f})^{(\mathtt{max}(\mathtt{d}_1,\mathtt{d}_2))}$ *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., $\forall \mathtt{f}^{(0)}: \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g}^{(2)}: \mathtt{Y} \to \mathtt{Z}$, $\mathtt{g} \circ \mathtt{f}: \mathtt{X} \to \mathtt{Z}$ 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 $\mathscr{L}_{dec}$, or translate the terms into the $\mathscr{L}_{dec}$ first, and then compose them there. Notice that the red colored composition symbol ($\circ$), 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 ($\mathscr{L}_{st}$)

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* ($\mathscr{L}_{st}$). 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 $\mathscr{L}_{st}$ where $\mathbb{1}$ is the singleton type while $\mathtt{V_i}$ 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 $\mathtt{X} \times \mathtt{Y}$ representing type products.

Terms are closed under composition ($\circ$) and pairing ($\langle \_,\_ \rangle_1$). I.e., for all terms $\mathtt{f}: \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g}: \mathtt{Y} \to \mathtt{Z}$, we have $\mathtt{g} \circ \mathtt{f}: \mathtt{X} \to \mathtt{Z}$. Similarly, for all $\mathtt{f}: \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g}: \mathtt{X} \to \mathtt{Z}$, there is $\langle \mathtt{f}, \mathtt{g} \rangle_1 : \mathtt{X} \to \mathtt{Y} \times \mathtt{Z}$. Notice that the pair subscript '1' denotes the left pairs. One can define in a symmetric way the right pairs for terms $\mathtt{f}: \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g}: \mathtt{X} \to \mathtt{Z}$ as $\langle \mathtt{f}, \mathtt{g} \rangle_r := \mathtt{permut} \circ \langle \mathtt{g}, \mathtt{f} \rangle_1$ where $\mathtt{permut} := \langle \pi_2, \pi_1 \rangle_1$. In the same way, one can respectively obtain left and right products of terms $\mathtt{f}: \mathtt{X_1} \to \mathtt{Y_1}$ and $\mathtt{g}: \mathtt{X_2} \to \mathtt{Y_2}$ as $\mathtt{f} \times_1 \mathtt{g} := \langle \mathtt{f} \circ \pi_1, \mathtt{g} \circ \pi_2 \rangle_1$ and $\mathtt{f} \times_r \mathtt{g} := \langle \mathtt{f} \circ \pi_1, \mathtt{g} \circ \pi_2 \rangle_r$. 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., $\forall \mathtt{f}^{(0)}: \mathtt{X} \to \mathtt{Y}$ and

Grammar of the decorated logic for the state: $(\mathtt{i} \in \mathtt{Loc})$

Types:	$\mathtt{t}, \mathtt{s}$	$::=$	$\mathtt{X} \mid \mathtt{Y} \mid \cdots \mid \mathtt{t} \times \mathtt{s} \mid \mathbb{1} \mid \mathtt{V_i}$
Decorations for terms:	$(\mathtt{d_1}), (\mathtt{d_2})$	$::=$	$(0) \mid (1) \mid (2)$
Terms:	$\mathtt{f}, \mathtt{g}$	$::=$	$\mathtt{a}^{(\mathtt{d})} \mid \mathtt{b}^{(\mathtt{d})} \mid \cdots \mid \mathtt{g} \circ \mathtt{f}^{(\mathtt{d})} \mid$ $\langle \mathtt{f}^{(\mathtt{d_1})} : \mathtt{X} \to \mathtt{Y}, \mathtt{g}^{(\mathtt{d_2})} : \mathtt{X} \to \mathtt{Z} \rangle_{\mathtt{l}}^{(\mathtt{max}(\mathtt{d_1},\mathtt{d_2}))} : \mathtt{X} \to \mathtt{Y} \times \mathtt{Z} \mid$ $\mathtt{lookup}_{\mathtt{i}}^{(1)} \mid \mathtt{update}_{\mathtt{i}}^{(2)} \mid (\mathtt{tpure}\ \bullet)^{(0)}$
Equations:	$\mathtt{eq}$	$::=$	$\mathtt{f}^{(\mathtt{d})} \equiv \mathtt{g}^{(\mathtt{d})} \mid \mathtt{f}^{(\mathtt{d})} \sim \mathtt{g}^{(\mathtt{d})}$

Figure 12: $\mathscr{L}_{st}$: syntax

$\mathtt{g}^{(2)} : \mathtt{X} \to \mathtt{Z}$, the term $\langle \mathtt{f}, \mathtt{g} \rangle_{\mathtt{l}} : \mathtt{X} \to \mathtt{Y} \times \mathtt{Z}$ takes the decoration (2). Note that in $\mathscr{L}_{st}$, we do not necessarily stick to the sequential products, even pairs/products of modifiers (intuitively parallel execution of modifiers) 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 `is_pair` of the inductive type `is`.

The interface terms are $\mathtt{lookup}_{\mathtt{i}} : \mathbb{1} \times \mathtt{S} \to \mathtt{V_i}$ and $\mathtt{update}_{\mathtt{i}} : \mathtt{V_i} \times \mathtt{S} \to \mathbb{1} \times \mathtt{S}$ where $\mathtt{S}$ denotes the distinguished object of states which never appears in the decorated setting. The use of decorations provides a new schema where term signatures are constructed without any occurrence of the state object. For instance, $\mathtt{lookup}_{\mathtt{i}}^{(1)} : \mathbb{1} \to \mathtt{V_i}$ is an accessor while $\mathtt{update}_{\mathtt{i}}^{(2)} : \mathtt{V_i} \to \mathbb{1}$ is a modifier. This way, we keep signatures close to their syntax and compose compatible terms as usual. The term `lookup` reads the value stored in a given location while `update` modifies it. We can call them *the unique sources of impurity*, since only the terms including `lookup` or `update` are impure; meaning those do not include them are pure with respect to the state effect.

The identity term `id`, the canonical pair projections π_1, π_2, the empty pair $\langle\ \rangle$ and `constants` are translated from a pure type system with type products using the `tpure` constructor, for all types `X` and `Y`, as follows:

$$
\begin{array}{ccccc}
\mathtt{id}_{\mathtt{X}}^{(0)} & : & \mathtt{X} \to \mathtt{X} & := & \mathtt{tpure}\ (\lambda\, \mathtt{x} : \mathtt{X}. \mathtt{x} : \mathtt{X}) \\
\pi_1^{(0)} & : & \mathtt{X} \times \mathtt{Y} \to \mathtt{X} & := & \mathtt{tpure}\ \mathtt{fst} \\
\pi_2^{(0)} & : & \mathtt{X} \times \mathtt{Y} \to \mathtt{Y} & := & \mathtt{tpure}\ \mathtt{snd} \\
\langle\ \rangle_{\mathtt{X}}^{(0)} & : & \mathtt{X} \to \mathbb{1} & := & \mathtt{tpure}\ (\lambda\ \mathtt{x} :\ \mathtt{X}.\ \mathtt{void} :\ \mathbb{1}) \\
\mathtt{constant}_{\mathtt{x}}^{(0)} & : & \mathbb{1} \to \mathtt{X} & := & \mathtt{tpure}\ (\lambda\ \_.\ \mathtt{x} : \mathtt{X})
\end{array}
$$

where `fst` and `snd` are constructors of product types.

The intended model of the above grammar is built with respect to the set of states `S` where a pure term $\mathtt{p}^{(0)} : \mathtt{X} \to \mathtt{Y}$ is interpreted as a function $\mathtt{p} : \mathtt{X} \to \mathtt{Y}$, an accessor $\mathtt{a}^{(1)} : \mathtt{X} \to \mathtt{Y}$ as a function $\mathtt{a} : \mathtt{X} \times \mathtt{S} \to \mathtt{Y}$, and a modifier $\mathtt{m}^{(2)} : \mathtt{X} \to \mathtt{Y}$ as a function $\mathtt{m} : \mathtt{X} \times \mathtt{S} \to \mathtt{Y} \times \mathtt{S}$. The complete and detailed category theoretical 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:

$$\text{(pwrepl)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}^{(0)} \colon \mathtt{Y} \to \mathtt{Z}}{\mathtt{g}^{(0)} \circ \mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{g}^{(0)} \circ \mathtt{f}_2^{(\mathtt{d}_2)}} \qquad \text{(wsubs)}\ \frac{\mathtt{g}^{(\mathtt{d}_3)} \colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{Y} \to \mathtt{Z}}{\mathtt{f}_1^{(\mathtt{d}_1)} \circ \mathtt{g}^{(\mathtt{d}_3)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \circ \mathtt{g}^{(\mathtt{d}_3)}}$$

$$\text{(replsubs)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)} \equiv \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{g}_1^{(\mathtt{d}_3)} \equiv \mathtt{g}_2^{(\mathtt{d}_4)} \colon \mathtt{Y} \to \mathtt{Z}}{\mathtt{g}_1^{(\mathtt{d}_3)} \circ \mathtt{f}_1^{(\mathtt{d}_1)} \equiv \mathtt{g}_2^{(\mathtt{d}_4)} \circ \mathtt{f}_2^{(\mathtt{d}_2)}} \qquad \text{(w\_unit)}\ \frac{\mathtt{f}^{(\mathtt{d}_1)} \colon \mathtt{X} \to \mathbb{1}}{\mathtt{f}^{(\mathtt{d}_1)} \sim \langle\ \rangle_{\mathtt{X}}^{(0)}}$$

$$(\mathrm{ax}_1)\ \frac{}{\mathtt{lookup}_{\mathtt{i}}^{(1)} \circ \mathtt{update}_{\mathtt{i}}^{(2)} \sim \mathtt{id}_{\mathtt{V}_\mathtt{i}}{}^{(0)}} \qquad (\mathrm{ax}_2)\ \frac{\forall \mathtt{i}, \mathtt{j} \in \mathtt{Loc},\ \mathtt{i} \neq \mathtt{j}}{\mathtt{lookup}_{\mathtt{i}}^{(1)} \circ \mathtt{update}_{\mathtt{j}}^{(2)} \sim \mathtt{lookup}_{\mathtt{i}}^{(1)} \circ \langle\ \rangle_{V_j}^{(0)}}$$

$$\text{(effect)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)}, \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \quad \langle\ \rangle_{\mathtt{Y}}^{(0)} \circ \mathtt{f}_1^{(\mathtt{d}_1)} \equiv \langle\ \rangle_{\mathtt{Y}}^{(0)} \circ \mathtt{f}_2^{(\mathtt{d}_2)}}{\mathtt{f}_1^{(\mathtt{d}_1)} \equiv \mathtt{f}_2^{(\mathtt{d}_2)}}$$

$$\text{(local\_global)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)}, \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathbb{1} \quad \forall\, \mathtt{i} \in \mathtt{Loc}, \mathtt{lookup}_{\mathtt{i}}^{(1)} \circ \mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{lookup}_{\mathtt{i}}^{(1)} \circ \mathtt{f}_2^{(\mathtt{d}_2)}}{\mathtt{f}_1^{(\mathtt{d}_1)} \equiv \mathtt{f}_2^{(\mathtt{d}_2)}}$$

$$\text{(w\_lpair\_eq)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)} \colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathtt{Z} \quad \mathtt{d}_1 \in \{0,1\}}{\pi_1^{(0)} \circ \langle \mathtt{f}_1, \mathtt{f}_2 \rangle_{\mathtt{l}}^{(\mathtt{max}(\mathtt{d}_1, \mathtt{d}_2))} \sim \mathtt{f}_1^{(\mathtt{d}_1)}}$$

$$\text{(s\_lpair\_eq)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)} \colon \mathtt{X} \to \mathtt{Y} \quad \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathtt{Z} \quad \mathtt{d}_1 \in \{0,1\}}{\pi_2^{(0)} \circ \langle \mathtt{f}_1, \mathtt{f}_2 \rangle_{\mathtt{l}}^{(\mathtt{max}(\mathtt{d}_1, \mathtt{d}_2))} \equiv \mathtt{f}_2^{(\mathtt{d}_2)}}$$

Figure 13: $\mathscr{L}_{st}$: 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 $\mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{X} \to \mathtt{Y}$ and a term $\mathtt{g} \colon \mathtt{Y} \to \mathtt{Z}$, it is possible to get the equation $\mathtt{g} \circ \mathtt{f}_1 \sim \mathtt{g} \circ \mathtt{f}_2$ only when the term $\mathtt{g}$ is pure. At this stage, we have no information about the modifications that $\mathtt{f}_1$ and $\mathtt{f}_2$ make on the memory state. Therefore, the post executed impure term $\mathtt{g}$ would destroy this result equality, for instance by reading the location $\mathtt{i}$ on which $\mathtt{f}_1$ and $\mathtt{f}_2$ has performed different modifications (pwrepl). However, the term substitution can be done regardless of the term decoration. I.e., given the equation $\mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{Y} \to \mathtt{Z}$ and a term $\mathtt{g}^{(\mathtt{d}_3)} \colon \mathtt{X} \to \mathtt{Y}$, it is possible to get the equation $\mathtt{f}_1 \circ \mathtt{g} \sim \mathtt{f}_2 \circ \mathtt{g}$ independent from the decoration of the term $\mathtt{g}$. We already now that $\mathtt{f}_1$ and $\mathtt{f}_2$ return the same result, executing any term $\mathtt{g}$ 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 $\mathtt{f} \colon \mathtt{X} \to \mathbb{1}$ with no result returned "`void`" (the unique inhabitant of $\mathbb{1}$ type) has an obvious result equality with the canonical empty pair $\langle\ \rangle_{\mathtt{X}}$ (w_unit).

The fundamental equations are given with the rules (ax_1) and (ax_2). The former states that by updating the location $\mathtt{i}$ with a value $\mathtt{v}$ and then observing the same location, one gets the value $\mathtt{v}$. This outputs the same value with the identity term $\mathtt{id}_{\mathtt{V}_\mathtt{i}}$, if it takes $\mathtt{v}$ as an argument. However, notice that these two ways of getting the value $\mathtt{v}$ have different state manipulations which makes them *weakly equal*. The latter, (ax_2), is to assume that updating the location $\mathtt{j}$ with a value $\mathtt{v}$ and then reading the content of a different location

$\mathtt{i}$ would return the same value with first throwing out the value $\mathtt{v}$ then observing the content of the location $\mathtt{i}$. They have different manipulations on the state so that they are *weakly equal*.

Two modifiers $\mathtt{f}_1^{(2)}, \mathtt{f}_2^{(2)} : \mathtt{X} \to \mathtt{Y}$ modify the state in the same way if and only if $\langle\,\rangle_{\mathtt{Y}} \circ \mathtt{f}_1 \equiv \langle\,\rangle_{\mathtt{Y}} \circ \mathtt{f}_2 : \mathtt{X} \to \mathbb{1}$, where $\langle\,\rangle_{\mathtt{Y}} : \mathtt{Y} \to \mathbb{1}$ throws out the returned value. So that $\mathtt{f}_1^{(2)}, \mathtt{f}_2^{(2)} : \mathtt{X} \to \mathtt{Y}$ are *strongly equal* if and only if $\mathtt{f}_1 \sim \mathtt{f}_2$ and $\langle\,\rangle_{\mathtt{Y}} \circ \mathtt{f}_1 \equiv \langle\,\rangle_{\mathtt{Y}} \circ \mathtt{f}_2$ (effect). Notice that this rule is valid also for the other decorations of terms $\mathtt{f}_1$ and $\mathtt{f}_2$.

Locally, the strong equality between two modifiers $\mathtt{f}_1^{(2)}, \mathtt{f}_2^{(2)} : \mathtt{X} \to \mathbb{1}$ can also be expressed as a pair of weak equations: $\mathtt{f}_1 \sim \mathtt{f}_2$ and $\forall \mathtt{i} : \mathtt{Loc}, \mathtt{lookup}_\mathtt{i} \circ \mathtt{f}_1 \sim \mathtt{lookup}_\mathtt{i} \circ \mathtt{f}_2$. The latter intuitively means that $\mathtt{f}_1$ and $\mathtt{f}_2$ 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 $\mathtt{f}_1 \sim \mathtt{f}_2$. It suffices to see whether $\forall \mathtt{i} : \mathtt{Loc}, \mathtt{lookup}_\mathtt{i} \circ \mathtt{f}_1 \sim \mathtt{lookup}_\mathtt{i} \circ \mathtt{f}_2$ to end up with $\mathtt{f}_1 \equiv \mathtt{f}_2$ (local_global). The rule is valid also for the other decorations of terms $\mathtt{f}_1$ and $\mathtt{f}_2$.

With (w_lpair_eq) and (w_rpair_eq) term pairs are characterized: the (left) pair structure $\langle \mathtt{f}_1, \mathtt{f}_2 \rangle_\mathtt{l}$ cannot be used when $\mathtt{f}_1$ and $\mathtt{f}_2$, both are modifiers, since it may lead to a conflict on the returned result. However, it can be used only when $\mathtt{f}_1$ is an accessor. We state by (w_lpair_eq) that $\langle \mathtt{f}_1, \mathtt{f}_2 \rangle_\mathtt{l}^{(\mathtt{max}(\mathtt{d}_1, \mathtt{d}_2))}$ has only result equality with $\mathtt{f}_1^{(\mathtt{d}_1)}$ and by (w_rpair_eq) that it has both result and effect equality with $\mathtt{f}_2^{(\mathtt{d}_2)}$.

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:

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

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

$(3)_d$ Interaction update-update. *Storing value the values* $\mathtt{x}$ *and* $\mathtt{y}$ *in a row to the same location* $\mathtt{i}$ *is just like storing* $\mathtt{y}$ *in it.* $\forall\, \mathtt{i} \in \mathtt{Loc}$, $\mathtt{update}_\mathtt{i}^{(2)} \circ \pi_2^{(0)} \circ (\mathtt{update}_\mathtt{i}^{(2)} \times_\mathtt{r} \mathtt{id}_{\mathtt{V}_\mathtt{i}}^{(0)}) \equiv \mathtt{update}_\mathtt{i}^{(2)} \circ \pi_2^{(0)} : \mathtt{V}_\mathtt{i} \times \mathtt{V}_\mathtt{i} \to \mathbb{1}$.

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

$(5)_d$ Commutation lookup-lookup. *The order of reading two different locations* $\mathtt{i}$ *and* $\mathtt{j}$ *does not matter.* $\forall\, \mathtt{i} \neq \mathtt{j} \in \mathtt{Loc}$, $(\mathtt{id}_{\mathtt{V}_\mathtt{i}}^{(0)} \times_\mathtt{r} \mathtt{lookup}_\mathtt{j}^{(1)}) \circ \pi_1^{-1(0)} \circ \mathtt{lookup}_\mathtt{i}^{(1)} \equiv \mathtt{permut}_{\mathtt{j},\mathtt{i}}^{(0)} \circ (\mathtt{id}_{\mathtt{V}_\mathtt{j}}^{(0)} \times_\mathtt{r} \mathtt{lookup}_\mathtt{i}^{(1)}) \circ \pi_1^{-1(0)} \circ \mathtt{lookup}_\mathtt{j}^{(1)} : \mathbb{1} \to \mathtt{V}_\mathtt{i} \times \mathtt{V}_\mathtt{j}$ where $\pi_1^{-1(0)} := \langle \mathtt{id}, \langle\,\rangle \rangle_\mathtt{l}^{(0)}$.

$(6)_d$ Commutation update-update. *The order of storing in two different locations* $\mathtt{i}$ *and* $\mathtt{j}$ *does not matter.* $\forall\, \mathtt{i} \neq \mathtt{j} \in \mathtt{Loc}$, $\mathtt{update}_\mathtt{j}^{(2)} \circ \pi_2^{(0)} \circ (\mathtt{update}_\mathtt{i}^{(2)} \times_\mathtt{r} \mathtt{id}_{\mathtt{V}_\mathtt{j}}^{(0)}) \equiv \mathtt{update}_\mathtt{i}^{(2)} \circ \pi_1^{(0)} \circ (\mathtt{id}_{\mathtt{V}_\mathtt{i}}^{(0)} \times_\mathtt{l} \mathtt{update}_\mathtt{j}^{(2)}) : \mathtt{V}_\mathtt{i} \times \mathtt{V}_\mathtt{j} \to \mathbb{1}$.

$(7)_d$ Commutation update-lookup. *The order of storing in a location* $\mathtt{i}$ *and reading another location* $\mathtt{j}$ *does not matter.* $\forall\, \mathtt{i} \neq \mathtt{j} \in \mathtt{Loc},\ \mathtt{lookup}_{\mathtt{j}}^{(1)} \circ \mathtt{update}_{\mathtt{i}}^{(2)} \equiv \pi_2^{(0)} \circ (\mathtt{update}_{\mathtt{i}}^{(2)} \times_{\mathtt{r}} \mathtt{id}_{\mathtt{V_j}}^{(0)}) \circ (\mathtt{id}_{\mathtt{V_i}}^{(0)} \times_{\mathtt{l}} \mathtt{lookup}_{\mathtt{j}}^{(1)}) \circ \pi_1^{-1(0)} : \mathtt{V_i} \to \mathtt{V_j}$.

$(8)_d$ Commutation lookup-constant. *Just after storing a constant* $\mathtt{c}$ *in a location* $\mathtt{i}$, *observing the content of* $\mathtt{i}$ *is the same as regenerating the constant* $\mathtt{c}$. $\forall\, \mathtt{i} \in \mathtt{Loc},\ \forall\, \mathtt{c} \in \mathtt{V_i};\ \mathtt{lookup}_{\mathtt{i}}^{(1)} \circ \mathtt{update}_{\mathtt{i}}^{(2)} \circ \mathtt{constant\,c}^{(0)} \equiv \mathtt{constant\,c}^{(0)} \circ \mathtt{update}_{\mathtt{i}}^{(2)} \circ \mathtt{constant\,c}^{(0)} : \mathbb{1} \to \mathtt{V_i}$.

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

4.2 $\mathscr{L}_{st}$ in Coq

In this section, we aim to highlight some crucial points of the $\mathscr{L}_{st}$ 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 `Loc : Type`. Since memory locations may contain different types of values, we also assume an arrow type `Val : Loc → Type` that is the type of values contained in each location. This fixes a type for every location. Note that the system thus does not support reasoning about *strong updates*.

```
Parameters (Loc: Type)  (Val: Loc → Type).
```

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

```
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 "o" ≜ comp (at level 70).
```

The constructor `tpure` takes a Coq side (pure) function and translates it into the decorated environment. The `comp` constructor deals with the composition of two compatible terms. I.e., given a pair of terms `f : term X Y` and `g : term Y Z`, then the composition `f ∘ g` would be an instance of the type `term X Z`. For the sake of conciseness, infix '∘' is used to denote the term composition. Similarly, the (left) `pair` constructor is to constitute pairs of compatible terms. I.e., given `f : term Y X` and `g : term Z X`, we have pair $\langle \mathtt{f},\mathtt{g}\rangle_{\mathtt{l}} : \mathtt{term}\ (\mathtt{Y} \times \mathtt{Z})\ \mathtt{X}$. Instead of the symbol $\langle \_,\_ \rangle_{\mathtt{l}}$, we use the keyword `pair` in the implementation. The terms `lookup` and `update` come as no surprise; just that the singleton type $\mathbb{1}$ and the type of values $\mathtt{V_i}$ are respectively called `unit` and `Val i` in the code. The terms such as the identity, the pair projections, the empty pair and the constant function can be derived from the native Coq functions with the use of `tpure` constructor as follows:

```
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 $\langle\,\rangle_{\mathtt{X}}$ from any type `X` to $\mathbb{1}$ 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 `term`s 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 → ∀ X Y, term X Y → Prop ≜
 | is_tpure: ∀ X Y (f: X → Y), is pure (@tpure X Y f)
 | is_comp: ∀ k X Y Z (f: term X Y) (g: term Y Z), is k f → is k g → is k (f o g)
 | is_pair: ∀ k X Y Z (f: term X Z) (g: term Y Z), is ro f → is k f → is k g → is k (pair f g)
 | is_lookup: ∀ i, is ro (lookup i)
 | is_update: ∀ i, is rw (update i)
 | is_pure_ro: ∀ X Y (f: term X Y), is pure f → is ro f
 | is_ro_rw: ∀ X Y  (f: term X Y), is ro f → is rw f.
```

Any term that is built by the `tpure` constructor is pure (`is_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_comp`), (`is_pure_ro`) and (`is_ro_rw`): see Lemma 3.2, and the corresponding Coq proof here [(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_pair`), (`is_pure_ro`) and (`is_ro_rw`). See the Coq proof of this fact here [(iv)]. We declare that the term `lookup` is an accessor (`is_lookup`), and the term `update` is a modifier (`is_update`). The last two constructors (`is_pure_ro`) and (`is_ro_rw`) define the decoration hierarchies.

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

```
Lemma is_pi1 X Y: is pure (@pi1 X Y).
Proof. apply is_tpure. Qed.
```

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

(iii) https://github.com/ekiciburak/decorated-logic-for-states-effect/blob/master/Decorations.v#L76-L79

(iv) https://github.com/ekiciburak/decorated-logic-for-states-effect/blob/master/Decorations.v#L81-L84 notation $==$ and $\sim$ to denote strong and weak equalities, respectively.

```
Definition idem X Y (x y: term X Y) ≜ x = y.
Inductive strong: ∀ X Y, relation (term X Y) ≜
| refl X Y: Reflexive (@strong X Y)
| sym: ∀ X Y, Symmetric (@strong X Y)
| trans: ∀ X Y, Transitive (@strong X Y)
| replsubs: ∀ X Y Z, Proper (@strong X Y ⟹ @strong Y Z ⟹ @strong X Z) comp
| ids: ∀ X Y (f: term X Y), f o id == f
| idt: ∀ X Y (f: term X Y), id o f == f
| assoc: ∀ X Y Z T (f: term X Y) (g: term Y Z) (h: term Z T), f o (g o h) == (f o g) o h
| wtos: ∀ X Y (f g: term X Y), is ro f → is ro g → f ~ g → f == g
| s_lpair_eq: ∀ X Y' Y (f1: term Y X) (f2: term Y' X), is ro f1 → pi2 o pair f1 f2 == f2
| effect: ∀ X Y (f g: term Y X), forget o f == forget o g → f ~ g → f == g
| local_global: ∀ X (f g: term unit X), (∀ i: Loc, lookup i o f ~ lookup i o g) → f == g
| tcomp: ∀ X Y Z (f: Z → Y) (g: Y → X), tpure (compose g f) == tpure g o tpure f
with weak: ∀ X Y, relation (term X Y) ≜
| wsym: ∀ X Y, Symmetric (@weak X Y)
| wtrans: ∀ X Y, Transitive (@weak X Y)
| pwrepl: ∀ A B C (g: term C B), (is pure g) →  Proper (@weak B A ⟹ @weak C A) (comp g)
| wsubs: ∀ A B C, Proper (@weak C B ⟹ @idem B A ⟹ @weak C A) comp
| stow: ∀  X Y (f g: term X Y), f == g → f ~ g
| w_lpair_eq: ∀ X Y' Y (f1: term Y X) (f2: term Y' X), is ro f1 → pi1 o pair f1 f2 ~ f1
| w_unit: ∀ X (f g: term unit X), f ~ g
| ax1: ∀ i, lookup i o update i ~ id
| ax2: ∀ i j, i ≠ j → lookup j o update i ~ lookup j o forget
  where "x == y" ≜ (strong x y) and "x ~ y" ≜ (weak x y).
```

The rule `tcomp` states that the `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 `compose`) and then apply `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 `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:

```
(**  Commutation update update **)
Theorem CUU: ∀ i j: Loc, i≠j →  update j o (pi2 o (rprod (update i) (@id (Val j)))) ==
               update i o (pi1 o (lprod (@id (Val i)) (update j))).
```

where

```
Definition permut {X Y}: term (X*Y) (Y*X) ≜ pair pi2 pi1.
Definition rpair {X Y Z} (f: term Y X) (g: term Z X): term (Y*Z) X ≜ permut o pair g f.
Definition lprod {X Y X' Y'} (f: term X X') (g: term Y Y'): term (X*Y) (X'*Y') ≜ pair (f o pi1) (g o pi2).
Definition rprod {X Y X' Y'} (f: term X X') (g: term Y Y') ≜ permut o pair (g o pi2) (f o pi1).
```

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 ($\mathscr{L}_{exc}$)

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* ($\mathscr{L}_{exc}$). 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* (or *exceptional values*). 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 ($\mathscr{L}_{st}$). Based on this nice duality, we build the logic $\mathscr{L}_{exc}$, and detail it in the following.

Grammar of the decorated logic for the exception: $(\mathtt{e} \in \mathtt{EName})$

Types:	$\mathtt{t}, \mathtt{s}$	::=	$\mathtt{X} \mid \mathtt{Y} \mid \cdots \mid \mathtt{t}+\mathtt{s} \mid \mathbb{O} \mid \mathtt{EV_e}$
Decoration for terms:	$(\mathtt{d_1}), (\mathtt{d_2})$	::=	$(0) \mid (1) \mid (2)$
Terms:	$\mathtt{f}, \mathtt{g}$	::=	$\mathtt{a}^{(\mathtt{d})} \mid \mathtt{b}^{(\mathtt{d})} \mid \cdots \mid \mathtt{g} \circ \mathtt{f}^{(\mathtt{d})} \mid$
			$[\mathtt{f}^{(\mathtt{d_1})} \colon \mathtt{X} \to \mathtt{Y} \mid \mathtt{g}^{(\mathtt{d_2})} \colon \mathtt{Z} \to \mathtt{Y}]_{\mathtt{l}}^{(\mathtt{max}(\mathtt{d_1},\mathtt{d_2}))} \colon \mathtt{X}+\mathtt{Z} \to \mathtt{Y} \mid$
			$\mathtt{tag}_e^{(1)} \mid \mathtt{untag}_e^{(2)} \mid (\downarrow \mathtt{f})^{(1)} \mid (\mathtt{tpure}\ \bullet)^{(0)}$
Equations:	$\mathtt{eq}$	::=	$\mathtt{f}^{(\mathtt{d})} \equiv \mathtt{g}^{(\mathtt{d})} \mid \mathtt{f}^{(\mathtt{d})} \sim \mathtt{g}^{(\mathtt{d})}$

Figure 14: $\mathscr{L}_{exc}$: syntax

Figure 14 shows the grammar of $\mathscr{L}_{exc}$ where $\mathbb{O}$ is the empty (uninhabited) type while $\mathtt{EV_e}$ is the type of parameters for each exception name $\mathtt{e}$. We assume that there is a finite set of exception names called $\mathtt{EName}$. Given types $\mathtt{X}$ and $\mathtt{Y}$, we have $\mathtt{X}+\mathtt{Y}$ denoting co-product (disjoint union or sum) types. Terms are closed under composition ($\circ$) and co-pairing ($[\_ \mid \_]_{\mathtt{l}}$). I.e., for all terms $\mathtt{f} \colon \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g} \colon \mathtt{Y} \to \mathtt{Z}$, we have $\mathtt{g} \circ \mathtt{f} \colon \mathtt{X} \to \mathtt{Z}$. Similarly, for all $\mathtt{f} \colon \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g} \colon \mathtt{Z} \to \mathtt{Y}$, there is $[\mathtt{f} \mid \mathtt{g}]_{\mathtt{l}} \colon \mathtt{X}+\mathtt{Z} \to \mathtt{Y}$. Notice that the co-pair subscript '$\mathtt{l}$' denotes the left co-pairs. One can define in a symmetric way the right co-pairs for terms $\mathtt{f} \colon \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g} \colon \mathtt{Z} \to \mathtt{Y}$ as $[\mathtt{f} \mid \mathtt{g}]_{\mathtt{r}} := [\mathtt{g}, \mathtt{f}]_{\mathtt{l}} \circ \mathtt{permut}$ where $\mathtt{permut} := [\mathtt{in_2} \mid \mathtt{in_1}]_{\mathtt{l}}$. Similarly, one can respectively obtain left and right co-products (sums) of terms $\mathtt{f} \colon \mathtt{X_1} \to \mathtt{Y_1}$ and $\mathtt{g} \colon \mathtt{X_2} \to \mathtt{Y_2}$ as $\mathtt{f} +_{\mathtt{l}} \mathtt{g} := [\mathtt{in_1} \circ \mathtt{f} \mid \mathtt{in_2} \circ \mathtt{g}]_{\mathtt{l}}$ and $\mathtt{f} +_{\mathtt{r}} \mathtt{g} := [\mathtt{in_1} \circ \mathtt{f} \mid \mathtt{in_2} \circ \mathtt{g}]_{\mathtt{r}}$. The decoration of a co-pair (co-product) depends on the decoration of its components, always taking the larger. I.e., $\forall \mathtt{f}^{(0)} \colon \mathtt{X} \to \mathtt{Z}$ and $\mathtt{g}^{(2)} \colon \mathtt{Y} \to \mathtt{Z}$, $[\mathtt{f} \mid \mathtt{g}]_{\mathtt{l}} \colon \mathtt{X}+\mathtt{Y} \to \mathtt{Z}$ takes the decoration (2). Being dual to the pairs in $\mathscr{L}_{st}$ (which impose an evaluation order), co-pairs in $\mathscr{L}_{exc}$ are used to have *case distinction* among terms. Co-pairs of catchers are allowed to be constructed in the logic $\mathscr{L}_{exc}$. 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 $\mathtt{tag_e} : \mathtt{EV_e} \to \mathbb{O}+\mathtt{E}$ and $\mathtt{untag_e} : \mathbb{O}+\mathtt{E} \to \mathtt{EV_e}+\mathtt{E}$ where $\mathtt{E}$ 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, $\mathtt{tag}_\mathtt{e}^{(1)} : \mathtt{EV_e} \to \mathbb{O}$ is a thrower while $\mathtt{untag}_\mathtt{e}^{(2)} : \mathbb{O} \to \mathtt{EV_e}$ is a catcher. This way, we keep signatures close to their syntax and compose compatible terms as usual. The term $\mathtt{tag_e}$ encapsulates an ordinary value with an exception of name $\mathtt{e}$ while the term $\mathtt{untag_e}$ recovers the value from the exceptional case.

The '$\downarrow$' 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 `id`, the canonical co-pair inclusions $\mathtt{in}_1$ and $\mathtt{in}_2$, and the empty co-pair $[\,]_\mathtt{X}$ (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 `tpure` constructor, for all types `X` and `Y`, as follows:

$$
\begin{array}{lcccc}
\mathtt{id}_\mathtt{X}^{(0)} & : & \mathtt{X} \to \mathtt{X} & := & \mathtt{tpure}\ (\lambda\, \mathtt{x} : \mathtt{X}.\mathtt{x} : \mathtt{X}) \\
\mathtt{in}_1^{(0)} & : & \mathtt{X} \to \mathtt{X}+\mathtt{Y} & := & \mathtt{tpure\ inl} \\
\mathtt{in}_2^{(0)} & : & \mathtt{Y} \to \mathtt{X}+\mathtt{Y} & := & \mathtt{tpure\ inr} \\
[\,]_\mathtt{X}^{(0)} & : & \mathbb{O} \to \mathtt{X} & := & \mathtt{tpure}\ (\lambda\, \_ : \mathbb{O}.\ \mathtt{x} : \mathtt{X})
\end{array}
$$

where `inl` and `inr` are constructors of sum types, and in the definition of $[\,]_\mathtt{X}$, `X` is assumed to be inhabited.

The intended model of the grammar of the logic $\mathscr{L}_{exc}$ is built with respect to the set of exceptions `E` where a pure term $\mathtt{p}^{(0)} : \mathtt{X} \to \mathtt{Y}$ is interpreted as a function $\mathtt{p} : \mathtt{X} \to \mathtt{Y}$, a propagator $\mathtt{pp}^{(1)} : \mathtt{X} \to \mathtt{Y}$ as a function $\mathtt{pp} : \mathtt{X} \to \mathtt{Y}+\mathtt{E}$, and a catcher $\mathtt{c}^{(2)} : \mathtt{X} \to \mathtt{Y}$ as a function $\mathtt{c} : \mathtt{X}+\mathtt{E} \to \mathtt{Y}+\mathtt{E}$. 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 $\mathtt{throw}_{\mathtt{Y},\mathtt{e}}^{(1)}$ is defined as:

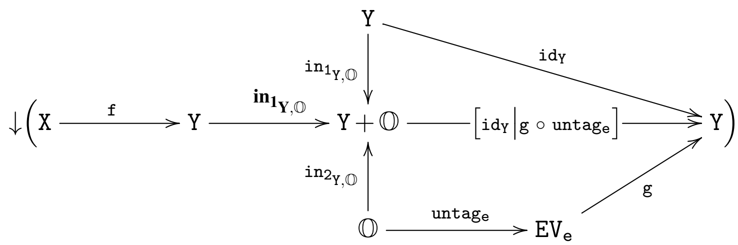
$$\mathtt{throw}_{\mathtt{Y},\mathtt{e}}^{(1)} := [\,]_\mathtt{Y}^{(0)} \circ \mathtt{tag}_\mathtt{e}^{(1)} : \mathtt{EV_e} \to \mathtt{Y}$$

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 $\mathtt{f}^{(1)} : \mathtt{X} \to \mathtt{Y}$, $\mathtt{g}^{(1)} : \mathtt{EV_e} \to \mathtt{Y}$ and each exception name `e`, the propagator $\mathtt{try(f)catch(e \Rightarrow g)}^{(1)}$ is defined in three steps, as follows:

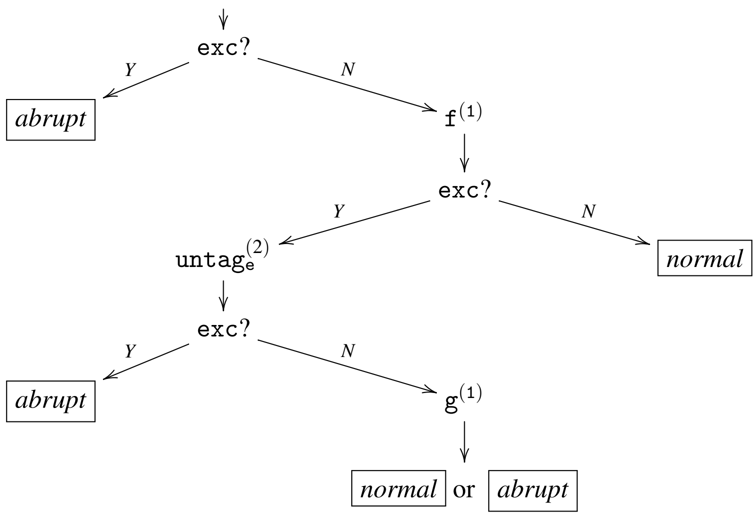
$$
\begin{array}{lcll}
\mathtt{Catch(e \Rightarrow g)}^{(2)} & := & [\ \mathtt{id}_\mathtt{Y}^{(0)} \mid \mathtt{g}^{(1)} \circ \mathtt{untag}_\mathtt{e}^{(2)}\ ]_1 & : \mathtt{Y}+\mathbb{O} \to \mathtt{Y} \\
\mathtt{Try(f)Catch(e \Rightarrow g)}^{(2)} & := & \mathtt{Catch(e \Rightarrow g)}^{(2)} \circ \mathtt{in}_{1_{\mathtt{Y},\mathbb{O}}}^{(0)} \circ \mathtt{f}^{(1)} & : \mathtt{X} \to \mathtt{Y} \\
\mathtt{try(f)catch(e \Rightarrow g)}^{(1)} & := & \downarrow\!\big(\mathtt{Try(f)Catch(e \Rightarrow g)}^{(2)}\big) & : \mathtt{X} \to \mathtt{Y}
\end{array}
$$

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



This, inside the downcast, intuitively tells us that if the term $\mathtt{f}$ throws an exception, then within the `Catch` block using the case distinction, provided by the copair, the exception is handled via the `untag` (unhandled exception gets propagated) and the continuation is the execution of the term $\mathtt{g}$. If $\mathtt{f}$ does not throw any exception then no handling is performed, we have `id` term in execution. Note also that the term $\mathtt{in1}_{\mathtt{Y},\mathbb{O}}$ used in the Definition 5.2 is the horizontal one in the above diagram, and implicitly means that $\mathtt{Y}$ and $\mathtt{Y}+\mathbb{O}$ isomorphic objects. The inclusions $\mathtt{in1}_{\mathtt{Y},\mathbb{O}}$ (the vertical in the above diagram) and $\mathtt{in2}_{\mathtt{Y},\mathbb{O}}$ play a role in the equational reasoning, given in Figure 15, that we provide on the top of $\mathscr{L}_{exc}$ syntax.

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



Remark 5.3. The decorated terms $\mathtt{throw}^{(1)}$ and $\mathtt{throw/catch}^{(1)}$ stated in Definitions 5.1 and 5.2 will serve, in Section 7 (see the translator function `dCmd`), as interpretations of the `IMP+Exc` commands `THROW` and `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 $\mathtt{f}_1^{(\mathtt{d}_1)} \sim \mathtt{f}_2^{(\mathtt{d}_2)} \colon \mathtt{Y} \to \mathtt{Z}$ and a term $\mathtt{g} \colon \mathtt{X} \to \mathtt{Y}$, it is possible to get the equation

Rules of the decorated logic for the exception:

$$\text{(pwsubs)}\frac{\mathtt{g}^{(0)}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_1^{(\mathtt{d}_1)}\sim\mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{Y}\to\mathtt{Z}}{\mathtt{f}_1^{(\mathtt{d}_1)}\circ\mathtt{g}^{(0)}\sim\mathtt{f}_2^{(\mathtt{d}_2)}\circ\mathtt{g}^{(0)}}\qquad \text{(wrepl)}\frac{\mathtt{f}_1^{(\mathtt{d}_1)}\sim\mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{X}\to\mathtt{Y}\quad \mathtt{g}^{(\mathtt{d}_3)}\colon\mathtt{Y}\to\mathtt{Z}}{\mathtt{g}^{(\mathtt{d}_3)}\circ\mathtt{f}_1^{(\mathtt{d}_1)}\sim\mathtt{g}^{(\mathtt{d}_3)}\circ\mathtt{f}_2^{(\mathtt{d}_2)}}$$

$$\text{(replsubs)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)}\equiv\mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{X}\to\mathtt{Y}\quad \mathtt{g}_1^{(\mathtt{d}_3)}\equiv\mathtt{g}_2^{(\mathtt{d}_4)}\colon\mathtt{Y}\to\mathtt{Z}}{\mathtt{g}_1^{(\mathtt{d}_3)}\circ\mathtt{f}_1^{(\mathtt{d}_1)}\equiv\mathtt{g}_2^{(\mathtt{d}_4)}\circ\mathtt{f}_2^{(\mathtt{d}_2)}}\qquad \text{(w\_empty)}\ \frac{\mathtt{f}^{(\mathtt{d}_1)}\colon\mathbb{O}\to\mathtt{X}}{\mathtt{f}^{(\mathtt{d}_1)}\sim[\,]_{\mathtt{X}}^{(0)}}$$

$$\text{(w\_downcast)}\ \frac{\mathtt{f}^{(2)}\colon\mathtt{Y}\to\mathtt{X}}{(\downarrow\mathtt{f})^{(1)}\sim\mathtt{f}^{(2)}}$$

$$\text{(eax}_1)\ \frac{}{\mathtt{untag}_{\mathtt{e}}^{(2)}\circ\mathtt{tag}_{\mathtt{e}}^{(1)}\sim\mathtt{id}_{\mathtt{EV_e}}^{(0)}}\qquad \text{(eax}_2)\ \frac{\forall\mathtt{e}_1,\mathtt{e}_2\in\mathtt{EName},\ \mathtt{e}_1\neq\mathtt{e}_2}{\mathtt{untag}_{\mathtt{e}_1}^{(2)}\circ\mathtt{tag}_{\mathtt{e}_2}^{(1)}\sim[\,]_{\mathtt{EV}_{\mathtt{e}_1}}^{(0)}\circ\mathtt{tag}_{\mathtt{e}_2}^{(1)}}$$

$$\text{(eeffect)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)},\mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{Y}\to\mathtt{X}\quad \mathtt{f}_1^{(\mathtt{d}_1)}\sim\mathtt{f}_2^{(\mathtt{d}_2)}\quad \mathtt{f}_1^{(\mathtt{d}_1)}\circ[\,]_{\mathtt{Y}}^{(0)}\equiv\mathtt{f}_2^{(\mathtt{d}_2)}\circ[\,]_{\mathtt{Y}}^{(0)}}{\mathtt{f}_1^{(\mathtt{d}_1)}\equiv\mathtt{f}_2^{(\mathtt{d}_2)}}$$

$$\text{(elocal\_global)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)},\mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathbb{O}\to\mathtt{X}\quad \forall\,\mathtt{e}\in\mathtt{EName},\ \mathtt{f}_1^{(\mathtt{d}_1)}\circ\mathtt{tag}_{\mathtt{e}}^{(1)}\sim\mathtt{f}_2^{(\mathtt{d}_2)}\circ\mathtt{tag}_{\mathtt{e}}^{(1)}}{\mathtt{f}_1^{(\mathtt{d}_1)}\equiv\mathtt{f}_2^{(\mathtt{d}_2)}}$$

$$\text{(w\_lcopair\_eq)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)}\colon\mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{Z}\to\mathtt{Y}\quad \mathtt{d}_1\in\{0,1\}}{[\mathtt{f}_1\mid\mathtt{f}_2]^{(\max(\mathtt{d}_1,\mathtt{d}_2))}\circ\mathtt{in}_1{}^{(0)}\sim\mathtt{f}_1^{(\mathtt{d}_1)}}$$

$$\text{(s\_lcopair\_eq)}\ \frac{\mathtt{f}_1^{(\mathtt{d}_1)}\colon\mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{Z}\to\mathtt{Y}\quad \mathtt{d}_1\in\{0,1\}}{[\mathtt{f}_1\mid\mathtt{f}_2]^{(\max(\mathtt{d}_1,\mathtt{d}_2))}\circ\mathtt{in}_2{}^{(0)}\equiv\mathtt{f}_2^{(\mathtt{d}_2)}}$$

Figure 15: $\mathscr{L}_{exc}$: rules

$\mathtt{f}_1\circ\mathtt{g}\sim\mathtt{f}_2\circ\mathtt{g}$ only when the term $\mathtt{g}$ is pure. At this stage, we have no information about the behaviors of $\mathtt{f}_1$ and $\mathtt{f}_2$ on exceptional values. Therefore, the pre-executed term $\mathtt{g}$ would destroy this result equality unless being pure, for instance, by throwing an exception of name $\mathtt{e}$ for which $\mathtt{f}_1$ and $\mathtt{f}_2$ 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 $\mathtt{f}_1^{(\mathtt{d}_1)}\sim\mathtt{f}_2^{(\mathtt{d}_2)}\colon\mathtt{X}\to\mathtt{Y}$ and a term $\mathtt{g}^{(\mathtt{d}_3)}\colon\mathtt{Y}\to\mathtt{Z}$, it is possible to get the equation $\mathtt{g}\circ\mathtt{f}_1\sim\mathtt{g}\circ\mathtt{f}_2$ independent from the decoration of the term $\mathtt{g}$. Since $\mathtt{f}_1$ and $\mathtt{f}_2$ behave the same on ordinary values, executing any term $\mathtt{g}$ after $\mathtt{f}_1$ and $\mathtt{f}_2$ 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 $\mathtt{f}\colon\mathbb{O}\to\mathtt{X}$ with no input parameter has an equivalence on ordinary values with the empty co-pair $[\,]_{\mathtt{X}}$ (w_empty). The rule (w_downcast) states that the term $(\downarrow\mathtt{f})$ behaves as $\mathtt{f}$, if the argument is ordinary. The fundamental equations are given with the rules (eax$_1$) and (eax$_2$). The former states that encapsulating an ordinary value with an exception of name $\mathtt{e}$ 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, (eax$_2$), is to assume that encapsulating an ordinary value $\mathtt{v}$ with an exception of name $\mathtt{e}_2$ and then trying to recover it from a different exception of name $\mathtt{e}_1$ would just lead $\mathtt{e}_2$ to be propagated. Similarly, if the ordinary value $\mathtt{v}$ is encapsulated with $\mathtt{e}_2$ with no recovery attempt afterwards would again lead $\mathtt{e}_2$ 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 $\mathtt{e}_1$) while right hand side propagates it.

Two catchers $\mathtt{f}_1^{(2)}, \mathtt{f}_2^{(2)} : \mathtt{X} \to \mathtt{Y}$ behave the same on exceptional values if and only if $\mathtt{f}_1 \circ [\,]_{\mathtt{X}} \equiv \mathtt{f}_2 \circ [\,]_{\mathtt{X}}$, where $[\,]_{\mathtt{X}} : \mathbb{O} \to \mathtt{X}$ throws out exceptional values. So that $\mathtt{f}_1^{(2)}, \mathtt{f}_2^{(2)} : \mathtt{X} \to \mathtt{Y}$ are *strongly equal* if and only if $\mathtt{f}_1 \sim \mathtt{f}_2$ and $\mathtt{f}_1 \circ [\,]_{\mathtt{X}} \equiv \mathtt{f}_2 \circ [\,]_{\mathtt{X}}$ (eeffect). The rule is valid also for the other decorations of terms $\mathtt{f}_1$ and $\mathtt{f}_2$.

Strong equality between two catchers $\mathtt{f}_1^{(2)}, \mathtt{f}_2^{(2)} : \mathbb{O} \to \mathtt{X}$ can also be expressed as a pair of weak equations: $\mathtt{f}_1 \sim \mathtt{f}_2$ and $\forall \mathtt{e} : \mathtt{ENname}, \mathtt{f}_1 \circ \mathtt{tag}_{\mathtt{e}} \sim \mathtt{f}_2 \circ \mathtt{tag}_{\mathtt{e}}$.The latter intuitively means that $\mathtt{f}_1$ and $\mathtt{f}_2$ behaves the same on all (finitely many) exceptional values when executed. Given that both behave the same on ordinary arguments (due to (w_empty)), there is no explicitly need to check if $\mathtt{f}_1 \sim \mathtt{f}_2$. It suffices to see whether $\forall \mathtt{e} : \mathtt{EName}, \mathtt{f}_1 \circ \mathtt{tag}_{\mathtt{e}} \sim \mathtt{f}_2 \circ \mathtt{tag}_{\mathtt{e}}$ to end up with $\mathtt{f}_1 \equiv \mathtt{f}_2$ (elocal_global). This rule is valid also for the other decorations of terms $\mathtt{f}_1$ and $\mathtt{f}_2$.

With (w_lcopair_eq) and (w_rcopair_eq), term co-pairs (sums) are characterized: the (left) co-pair structure $[\mathtt{f}_1 \mid \mathtt{f}_2]_1$ cannot be used when $\mathtt{f}_1$ and $\mathtt{f}_2$, both are catchers, since it may lead to a conflict on exceptional values. When $\mathtt{f}_1$ is a propagator, with (w-copair-eq), we assume that ordinary values of type $\mathtt{X}$ are treated by $[\mathtt{f}_1 \mid \mathtt{f}_2]_1^{(\mathtt{max}(\mathtt{d}_1,\mathtt{d}_2))}$ as they would be by $\mathtt{f}_1^{(\mathtt{d}_1)}$ and with (s-copair-eq) that ordinary values of type $\mathtt{Z}$ and exceptional values are treated by $[\mathtt{f}_1 \mid \mathtt{f}_2]_1^{(\mathtt{max}(\mathtt{d}_1,\mathtt{d}_2))}$ as they would be by $\mathtt{f}_2^{(\mathtt{d}_2)}$.

Similar to the ones of the logic $\mathscr{L}_{st}$, the rules of the logic $\mathscr{L}_{exc}$ 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:

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

$(2)_d$ Commutation untag-untag. *Untagging two distinct exception names can be done in any order.*
$\forall \mathtt{e} \neq \mathtt{r} \in \mathtt{EName},\ (\mathtt{untag}_{\mathtt{e}} +_{\mathtt{r}} \mathtt{id}_{\mathtt{EV}_{\mathtt{r}}})^{(2)} \circ \mathtt{in}_2^{(0)} \circ \mathtt{untag}_{\mathtt{r}}^{(2)} \equiv$
$(\mathtt{id}_{\mathtt{EV}_{\mathtt{e}}} +_{1} \mathtt{untag}_{\mathtt{r}})^{(2)} \circ \mathtt{in}_1^{(0)} \circ \mathtt{untag}_{\mathtt{e}}^{(2)} : \mathbb{O} \to \mathtt{EV}_{\mathtt{e}} + \mathtt{EV}_{\mathtt{r}}$.

$(3)_d$ Propagator-propagates. *A propagator term always propagates the exception.*
$\forall \mathtt{e} \in \mathtt{EName},\ \mathtt{a}^{(1)} : \mathtt{X} \to \mathtt{Y},\ \ \mathtt{a}^{(1)} \circ [\,]_{\mathtt{X}}^{(0)} \circ \mathtt{tag}_{\mathtt{e}}^{(1)} \equiv [\,]_{\mathtt{Y}}^{(0)} \circ \mathtt{tag}_{\mathtt{e}}^{(1)} : \mathtt{EV}_{\mathtt{e}} \to \mathtt{Y}$.

$(4)_d$ Recovery. *The parameter used for throwing an exception may be recovered.*
$\left(\forall \mathtt{f}^{(1)}, \mathtt{g}^{(1)} : \mathtt{X} \to \mathbb{O},\ \ [\,]_{\mathtt{Y}}^{(0)} \circ \mathtt{f}^{(1)} \equiv [\,]_{\mathtt{Y}}^{(0)} \circ \mathtt{g}^{(1)} \implies \mathtt{f}^{(1)} \equiv \mathtt{g}^{(1)}\right) \implies$
$\left(\forall \mathtt{e} \in \mathtt{EName},\ \mathtt{u}_1^{(0)}, \mathtt{u}_2^{(0)} : \mathtt{X} \to \mathtt{EV}_{\mathtt{e}}, \left(\mathtt{throw}_{\mathtt{e}}^{(1)} \circ \mathtt{u}_1^{(0)} \equiv \mathtt{throw}_{\mathtt{e}}^{(1)} \circ \mathtt{u}_2^{(0)}\right) \implies \mathtt{u}_1^{(0)} \equiv \mathtt{u}_2^{(0)} : \mathtt{X} \to \mathtt{EV}_{\mathtt{e}}\right)$.

$(5)_d$ Try. *The strong equation is compatible with* `try/catch`.
$\forall \mathtt{e} \in \mathtt{EName},\ \mathtt{a}_1^{(1)}, \mathtt{a}_2^{(1)} : \mathtt{X} \to \mathtt{Y}, \mathtt{b}^{(1)} : \mathtt{EV}_{\mathtt{e}} \to \mathtt{Y},\ \mathtt{a}_1^{(1)} \equiv \mathtt{a}_2^{(1)} \implies$
$\mathtt{try}(\mathtt{a}_1)\mathtt{catch}(\mathtt{e} \Rightarrow \mathtt{b})^{(1)} \equiv \mathtt{try}(\mathtt{a}_2)\mathtt{catch}(\mathtt{e} \Rightarrow \mathtt{b})^{(1)} : \mathtt{X} \to \mathtt{Y}$.

$(6)_d$ Try$_0$. *Pure code inside* `try` *never triggers the code inside* `catch`.
$\forall \mathtt{e} \in \mathtt{EName},\ \mathtt{u}^{(0)}: \mathtt{X} \to \mathtt{Y}, \mathtt{b}^{(1)}: \mathtt{EV_e} \to \mathtt{Y}, \mathtt{try(u)catch(e} \Rightarrow \mathtt{b)}^{(1)} \equiv \mathtt{u}^{(0)}: \mathtt{X} \to \mathtt{Y}.$

$(7)_d$ Try$_1$. *The code inside* `catch` *is executed as soon as an exception is thrown inside* `try`.
$\forall \mathtt{e} \in \mathtt{EName},\ \mathtt{u}^{(0)}: \mathtt{X} \to \mathtt{EV_e}, \mathtt{b}^{(1)}: \mathtt{EV_e} \to \mathtt{Y}, \mathtt{try(throw_e} \circ \mathtt{u)catch(e} \Rightarrow \mathtt{b)}^{(1)} \equiv \mathtt{b}^{(1)} \circ \mathtt{u}^{(0)}: \mathtt{X} \to \mathtt{Y}.$

$(8)_d$ Try$_2$. *An exception gets propagated, if the exception name is not pattern matched in* `catch`.
$\forall\, (\mathtt{e} \neq \mathtt{f}) \in \mathtt{EName},\ \mathtt{u}^{(0)}: \mathtt{X} \to \mathtt{EV_f}, \mathtt{b}^{(1)}: \mathtt{EV_e} \to \mathtt{Y},$
$\mathtt{try(throw_f} \circ \mathtt{u)catch(e} \Rightarrow \mathtt{b)}^{(1)} \equiv \mathtt{throw}_\mathtt{f}^{(1)} \circ \mathtt{u}^{(0)}: \mathtt{X} \to \mathtt{Y}.$

These are the archetype properties that we have proved within the scope of the $\mathscr{L}_{exc}$. To see these proofs, check out (Ekici, 2015, §6.7). Besides, we have implemented the $\mathscr{L}_{exc}$ 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., $\mathtt{f}^{(1)}$ and $\mathtt{g}^{(1)}$) 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 $\mathscr{L}_{exc}$ in Coq

Coq implementation of $\mathscr{L}_{exc}$ follows the same approach with the one for $\mathscr{L}_{st}$ as summarized in Section 4.2. We represent the set of exception names by a Coq parameter `EName : Type`. An arrow type `EVal : EName → 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 `epure`, `ppg` and `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:

```
Definition throw (X: Type) (e: EName)  ≜ (@empty X) o tag e.
Definition try_catch (X Y: Type) (e: EName) (f: term Y X) (g: term Y (Val e)) ≜
          downcast (copair (@id Y) (g o untag e) o in1 o f).
```

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:

```
(** Propagator propagates **)
Lemma PPT: ∀ X Y (e: EName) (a: term Y X), is ppg a →  a o ((@empty X) o tag e) == (@empty Y) o tag e.
```

The full Coq proofs of such properties can be found here [(viii)], and the entire implementation there[(ix)].

6 Combining $\mathscr{L}_{st}$ and $\mathscr{L}_{exc}$

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 $\mathscr{L}_{st}$ and $\mathscr{L}_{exc}$. We call the newly born logic *the decorated logic for the state and the exception*, and denote it $\mathscr{L}_{st+exc}$. To start with, we give the syntax of $\mathscr{L}_{st+exc}$ below in Figure 16.

Grammar of the decorated logic for the state and the exception: $(\mathtt{i} \in \mathtt{Loc})$ $(\mathtt{e} \in \mathtt{EName})$

Types:	$\mathtt{t},\mathtt{s}$	::=	$\mathtt{X} \mid \mathtt{Y} \mid \cdots \mid \mathtt{t}\times\mathtt{s} \mid \mathtt{t}+\mathtt{s} \mid \mathbb{1} \mid \mathbb{O} \mid \mathtt{V_i} \mid \mathtt{EV_e}$
Decoration for terms:	$(\mathtt{d_1},\mathtt{d_2}),(\mathtt{d_3},\mathtt{d_4})$	::=	$(0,0) \mid (0,1) \mid (0,2) \mid (1,0) \mid (1,1) \mid$ $(1,2) \mid (2,0) \mid (2,1) \mid (2,2)$
Terms:	$\mathtt{f},\mathtt{g}$	::=	$\mathtt{a}^{(\mathtt{d_1},\mathtt{d_2})} \mid \mathtt{b}^{(\mathtt{d_1},\mathtt{d_2})} \mid \cdots \mid \mathtt{g}\circ\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \mid$ $\langle \mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})},\mathtt{g}^{(\mathtt{d_3},\mathtt{d_4})}\rangle_{\mathtt{l}}^{(\mathtt{max}(\mathtt{d_1},\mathtt{d_3}),\mathtt{max}(\mathtt{d_2},\mathtt{d_4}))} \mid$ $[\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \mid \mathtt{g}^{(\mathtt{d_3},\mathtt{d_4})}]_{\mathtt{l}}^{(\mathtt{max}(\mathtt{d_1},\mathtt{d_3}),\mathtt{max}(\mathtt{d_2},\mathtt{d_4}))} \mid$ $\mathtt{lookup}_{\mathtt{i}}^{(1,0)} \mid \mathtt{update}_{\mathtt{i}}^{(2,0)} \mid \mathtt{tag}_{\mathtt{e}}^{(0,1)} \mid \mathtt{untag}_{\mathtt{e}}^{(0,2)} \mid$ $(\downarrow\ \mathtt{f})^{(0,1)} \mid (\mathtt{tpure}\ \bullet)^{(0,0)}$
Equations:	$\mathtt{eq}$	::=	$\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{g}^{(\mathtt{d_1},\mathtt{d_2})} \mid \mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \equiv\sim \mathtt{g}^{(\mathtt{d_1},\mathtt{d_2})} \mid$ $\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \mathtt{g}^{(\mathtt{d_1},\mathtt{d_2})} \mid \mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \sim\sim \mathtt{g}^{(\mathtt{d_1},\mathtt{d_2})}$

Figure 16: $\mathscr{L}_{st+exc}$: 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., $\mathtt{f}^{(1,2)}$ says that $\mathtt{f}$ 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., $\forall \mathtt{f}^{(1,2)}: \mathtt{X} \to \mathtt{Y}$ and $\mathtt{g}^{(2,1)}: \mathtt{Y} \to \mathtt{Z}$, $\mathtt{g}\circ\mathtt{f}: \mathtt{X} \to \mathtt{Z}$ takes the decoration $(2,2)$. The pairs/products of compatible terms $\mathtt{f}_1^{(2,2)}$, $\mathtt{g}_1^{(2,2)}$, and similarly the co-pair/co-products of compatible terms $\mathtt{f}_2^{(2,2)}$, $\mathtt{g}_2^{(2,2)}$ can be constructed within the scope of $\mathscr{L}_{st+exc}$ but cannot be used in the provided equational reasoning. This is because, $\mathtt{f}_1^{(2,2)}$ and $\mathtt{g}_1^{(2,2)}$, 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 $\mathtt{f}_2^{(2,2)}$ and $\mathtt{g}_2^{(2,2)}$, 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_lpair_eq), (s_lpair_eq), (w_lcopair_eq) and (s_lcopair_eq), in Figure 17, enforce these restrictions.

The types of $\mathscr{L}_{st+exc}$ is the union of the types of $\mathscr{L}_{st}$ and $\mathscr{L}_{exc}$. Similarly, the terms of $\mathscr{L}_{st+exc}$ is the union of the terms of $\mathscr{L}_{st}$ and $\mathscr{L}_{exc}$. The interface terms for the state effect are pure with respect to the exception and vice versa: $\mathtt{lookup}^{(1,0)}$, $\mathtt{update}^{(2,0)}$, $\mathtt{tag}^{(0,1)}$ and $\mathtt{untag}^{(0,2)}$. 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 π_1, π_2, the empty pair $\langle\ \rangle$, the canonical co-pair inclusions $\mathtt{in_1}$, $\mathtt{in_2}$, the empty co-pair $[\,]$ and `constants` from a pure type system with product and sum types using the `tpure` constructor, for all types `X` and `Y`, as:

$$\mathtt{id}_{\mathtt{X}}^{(0,0)} \quad : \quad \mathtt{X} \to \mathtt{X} \quad := \quad \mathtt{tpure}\ (\lambda\, \mathtt{x} : \mathtt{X}.\mathtt{x} : \mathtt{X})$$

$$\pi_1^{(0,0)} \quad : \quad \mathtt{X}\times\mathtt{Y} \to \mathtt{X} \quad := \quad \mathtt{tpure\,fst}$$

$$
\begin{array}{llcll}
\pi_2^{(0,0)} & : & \mathtt{X} \times \mathtt{Y} \to \mathtt{Y} & := & \mathtt{tpure\ snd} \\
\langle\,\rangle_{\mathtt{X}}^{(0,0)} & : & \mathtt{X} \to \mathbb{1} & := & \mathtt{tpure}\ (\lambda\ \mathtt{x} : \mathtt{X}.\ \mathtt{void} : \mathbb{1}) \\
\mathtt{in}_1^{(0,0)} & : & \mathtt{X} \to \mathtt{X}+\mathtt{Y} & := & \mathtt{tpure\ inl} \\
\mathtt{in}_2^{(0,0)} & : & \mathtt{Y} \to \mathtt{X}+\mathtt{Y} & := & \mathtt{tpure\ inr} \\
[\,]_{\mathtt{X}}^{(0,0)} & : & \mathbb{0} \to \mathtt{X} & := & \mathtt{tpure}\ (\lambda\ \_ : \mathbb{0}.\ \mathtt{x} : \mathtt{X}) \\
\mathtt{constant}_{\mathtt{x}}^{(0,0)} & : & \mathbb{1} \to \mathtt{X} & := & \mathtt{tpure}\ (\lambda\ \_.\ \mathtt{x} : \mathtt{X})
\end{array}
$$

where `fst` and `snd` are constructors of product types while `inl` and `inr` are of sum types, and in the definition of $[\,]_{\mathtt{X}}$, 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., $\mathtt{f}^{(0,2)}$ freely converts into $\mathtt{f}^{(1,2)}$. See all cases below:
  - $\dfrac{\mathtt{f}^{(0,\mathtt{d})}}{\mathtt{f}^{(1,\mathtt{d})}}, \quad \dfrac{\mathtt{f}^{(1,\mathtt{d})}}{\mathtt{f}^{(2,\mathtt{d})}}, \quad \dfrac{\mathtt{f}^{(\mathtt{d},0)}}{\mathtt{f}^{(\mathtt{d},1)}}, \quad \dfrac{\mathtt{f}^{(\mathtt{d},1)}}{\mathtt{f}^{(\mathtt{d},2)}}$ for $\mathtt{d} \in \{0,1,2\}$
- 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., $\mathtt{f} \equiv\sim \mathtt{g}$ 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.
  - $(\equiv\equiv\text{-to-}\equiv\sim)\dfrac{\mathtt{f}^{(2,2)} \equiv\equiv \mathtt{g}^{(2,2)}}{\mathtt{f}^{(2,2)} \equiv\sim \mathtt{g}^{(2,2)}}, \quad (\equiv\equiv\text{-to-}\sim\equiv)\dfrac{\mathtt{f}^{(2,2)} \equiv\equiv \mathtt{g}^{(2,2)}}{\mathtt{f}^{(2,2)} \sim\equiv \mathtt{g}^{(2,2)}}$
  - $(\equiv\sim\text{-to-}\sim\sim)\dfrac{\mathtt{f}^{(2,2)} \equiv\sim \mathtt{g}^{(2,2)}}{\mathtt{f}^{(2,2)} \sim\sim \mathtt{g}^{(2,2)}}, \quad (\sim\equiv\text{-to-}\sim\sim)\dfrac{\mathtt{f}^{(2,2)} \sim\equiv \mathtt{g}^{(2,2)}}{\mathtt{f}^{(2,2)} \sim\sim \mathtt{g}^{(2,2)}}$
  - $(\sim\equiv\text{-to-}\equiv\equiv)\dfrac{\mathtt{f}^{(1,2)} \sim\equiv \mathtt{g}^{(1,2)}}{\mathtt{f}^{(1,2)} \equiv\equiv \mathtt{g}^{(1,2)}}, \quad (\equiv\sim\text{-to-}\equiv\equiv)\dfrac{\mathtt{f}^{(2,1)} \equiv\sim \mathtt{g}^{(2,1)}}{\mathtt{f}^{(2,1)} \equiv\equiv \mathtt{g}^{(2,1)}}$
  - $(\sim\sim\text{-to-}\equiv\sim)\dfrac{\mathtt{f}^{(1,2)} \sim\sim \mathtt{g}^{(1,2)}}{\mathtt{f}^{(1,2)} \equiv\sim \mathtt{g}^{(1,2)}}, \quad (\sim\sim\text{-to-}\sim\equiv)\dfrac{\mathtt{f}^{(2,1)} \sim\sim \mathtt{g}^{(2,1)}}{\mathtt{f}^{(2,1)} \sim\equiv \mathtt{g}^{(2,1)}}$
- The rules of the logic $\mathscr{L}_{st+exc}$ 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 replay the whole rule bodies, and implicitly assume that all equality sorts are equivalence relations respecting the properties *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) $$\frac{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{g}^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{Y}\to\mathtt{Z}\quad \mathtt{h}^{(\mathtt{d_5},\mathtt{d_6})}\colon \mathtt{Z}\to\mathtt{T}}{\mathtt{h}^{(\mathtt{d_5},\mathtt{d_6})}\circ(\mathtt{g}^{(\mathtt{d_3},\mathtt{d_4})}\circ\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}) \equiv\equiv (\mathtt{h}^{(\mathtt{d_5},\mathtt{d_6})}\circ\mathtt{g}^{(\mathtt{d_3},\mathtt{d_4})})\circ\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}}$$

(ids) $$\frac{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}\colon \mathtt{X}\to\mathtt{Y}}{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}\circ\mathtt{id}_\mathtt{X}^{(0,0)} \equiv\equiv \mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}}$$ (idt) $$\frac{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}\colon \mathtt{X}\to\mathtt{Y}}{\mathtt{id}_\mathtt{Y}^{(0,0)}\circ\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}}$$

(pwrepl) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{g}^{(0,\mathtt{d_5})}\colon \mathtt{Y}\to\mathtt{Z}}{\mathtt{g}^{(0,\mathtt{d_5})}\circ\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \mathtt{g}^{(0,\mathtt{d_5})}\circ\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$ (wsubs) $$\frac{\mathtt{g}^{(\mathtt{d_5},\mathtt{d_6})}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{Y}\to\mathtt{Z}}{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}\circ\mathtt{g}^{(\mathtt{d_5},\mathtt{d_6})} \sim\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\circ\mathtt{g}^{(\mathtt{d_5},\mathtt{d_6})}}$$

(pwsubs) $$\frac{\mathtt{g}^{(\mathtt{d_5},0)}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\sim \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{Y}\to\mathtt{Z}}{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}\circ\mathtt{g}^{(\mathtt{d_5},0)} \equiv\sim \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\circ\mathtt{g}^{(\mathtt{d_5},0)}}$$ (wrepl) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\sim \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{g}^{(\mathtt{d_5},\mathtt{d_6})}\colon \mathtt{Y}\to\mathtt{Z}}{\mathtt{g}^{(\mathtt{d_5},\mathtt{d_6})}\circ\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\sim \mathtt{g}^{(\mathtt{d_5},\mathtt{d_6})}\circ\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(replsubs) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{g}_1^{(\mathtt{d_5},\mathtt{d_6})} \equiv\equiv \mathtt{g}_2^{(\mathtt{d_7},\mathtt{d_8})}\colon \mathtt{Y}\to\mathtt{Z}}{\mathtt{g}_1^{(\mathtt{d_5},\mathtt{d_6})}\circ\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{g}_2^{(\mathtt{d_7},\mathtt{d_8})}\circ\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(w_unit) $$\frac{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}\colon \mathtt{X}\to\mathbb{1}}{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \langle\,\rangle_\mathtt{X}^{(0,0)}}$$ (w_empty) $$\frac{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})}\colon \mathbb{0}\to\mathtt{X}}{\mathtt{f}^{(\mathtt{d_1},\mathtt{d_2})} \equiv\sim [\,]_\mathtt{X}^{(0,0)}}$$ (w_downcast) $$\frac{\mathtt{f}^{(\mathtt{d_1},2)}\colon \mathtt{Y}\to\mathtt{X}}{(\downarrow\mathtt{f})^{(\mathtt{d_1},1)} \equiv\sim \mathtt{f}^{(\mathtt{d_1},2)}}$$

(ax$_1$) $$\frac{}{\mathtt{lookup}_\mathtt{i}^{(1,0)}\circ\mathtt{update}_\mathtt{i}^{(2,0)} \sim\equiv \mathtt{id}_{\mathtt{V_i}}^{(0,0)}}$$ (ax$_2$) $$\frac{\forall \mathtt{i},\mathtt{j}\in\mathtt{Loc},\ \mathtt{i}\neq\mathtt{j}}{\mathtt{lookup}_\mathtt{i}^{(1,0)}\circ\mathtt{update}_\mathtt{j}^{(2,0)} \sim\equiv \mathtt{lookup}_\mathtt{i}^{(1,0)}\circ\langle\,\rangle_{V_i}^{(0,0)}}$$

(eax$_1$) $$\frac{}{\mathtt{untag}_\mathtt{e}^{(0,2)}\circ\mathtt{tag}_\mathtt{e}^{(0,1)} \equiv\sim \mathtt{id}_{\mathtt{EV_e}}^{(0,0)}}$$ (eax$_2$) $$\frac{\forall \mathtt{e_1},\mathtt{e_2}\in\mathtt{EName},\ \mathtt{e_1}\neq\mathtt{e_2}}{\mathtt{untag}_{\mathtt{e_1}}^{(0,2)}\circ\mathtt{tag}_{\mathtt{e_2}}^{(0,1)} \equiv\sim [\,]_{\mathtt{EV_e}}^{(0,0)}\circ\mathtt{tag}_{\mathtt{e_2}}^{(0,1)}}$$

(effect) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})},\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\quad \langle\,\rangle_\mathtt{Y}^{(0,0)}\circ\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \langle\,\rangle_\mathtt{Y}^{(0,0)}\circ\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(eeffect) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})},\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{Y}\to\mathtt{X}\quad \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\sim \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\quad \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}\circ[\,]_\mathtt{Y}^{(0,0)} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\circ[\,]_\mathtt{Y}^{(0,0)}}{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(local_global) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})},\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathtt{X}\to\mathbb{1}\quad \forall\,\mathtt{i}\in\mathtt{Loc},\ \mathtt{lookup}_\mathtt{i}^{(1,0)}\circ\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \sim\equiv \mathtt{lookup}_\mathtt{i}^{(1,0)}\circ\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(elocal_global) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})},\mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\colon \mathbb{0}\to\mathtt{X}\quad \forall\,\mathtt{e}\in\mathtt{EName},\ \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}\circ\mathtt{tag}_\mathtt{e}^{(0,1)} \equiv\sim \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}\circ\mathtt{tag}_\mathtt{e}^{(0,1)}}{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(w_lpair_eq) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}:\mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}:\mathtt{X}\to\mathtt{Z}\quad \mathtt{d_1}\in\{0,1\}}{\pi_1^{(0,0)}\circ\langle\mathtt{f_1},\mathtt{f_2}\rangle_1^{((\mathtt{max}(\mathtt{d_1},\mathtt{d_3}),\mathtt{max}(\mathtt{d_2},\mathtt{d_4}))} \sim\equiv \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}}$$

(s_lpair_eq) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}:\mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}:\mathtt{X}\to\mathtt{Z}\quad \mathtt{d_1}\in\{0,1\}}{\pi_2^{(0,0)}\circ\langle\mathtt{f_1},\mathtt{f_2}\rangle_1^{((\mathtt{max}(\mathtt{d_1},\mathtt{d_3}),\mathtt{max}(\mathtt{d_2},\mathtt{d_4}))} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

(w_lcopair_eq) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}:\mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}:\mathtt{Z}\to\mathtt{Y}\quad \mathtt{d_2}\in\{0,1\}}{[\mathtt{f_1}\mid\mathtt{f_2}]^{((\mathtt{max}(\mathtt{d_1},\mathtt{d_3}),\mathtt{max}(\mathtt{d_2},\mathtt{d_4}))}\circ\mathtt{in_1}^{(0,0)} \equiv\sim \mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}}$$

(s_lcopair_eq) $$\frac{\mathtt{f}_1^{(\mathtt{d_1},\mathtt{d_2})}:\mathtt{X}\to\mathtt{Y}\quad \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}:\mathtt{Z}\to\mathtt{Y}\quad \mathtt{d_2}\in\{0,1\}}{[\mathtt{f_1}\mid\mathtt{f_2}]^{((\mathtt{max}(\mathtt{d_1},\mathtt{d_3}),\mathtt{max}(\mathtt{d_2},\mathtt{d_4}))}\circ\mathtt{in_2}^{(0,0)} \equiv\equiv \mathtt{f}_2^{(\mathtt{d_3},\mathtt{d_4})}}$$

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

Figure 17: $\mathscr{L}_{st+exc}$: 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 $\equiv\equiv$. I.e., the statements `propagator-propagates` and `update-update` look like:

$$\forall \mathtt{e} \in \mathtt{EName},\ \mathtt{a}^{(0,1)} : \mathtt{X} \to \mathtt{Y},\ \ \mathtt{a}^{(0,1)} \circ [\,]_{\mathtt{X}}^{(0,0)} \circ \mathtt{tag}_{\mathtt{e}}^{(0,1)} \equiv\equiv [\,]_{\mathtt{Y}}^{(0,0)} \circ \mathtt{tag}_{\mathtt{e}}^{(0,1)} : \mathtt{EV}_{\mathtt{e}} \to \mathtt{Y}.$$

$$\forall \mathtt{i} \neq \mathtt{j} \in \mathtt{Loc},\ \mathtt{update}_{\mathtt{j}}^{(2,0)} \circ \pi_2^{(0,0)} \circ (\mathtt{update}_{\mathtt{i}}^{(2,0)} \times_{\mathtt{r}} \mathtt{id}_{\mathtt{V_j}}^{(0,0)}) \equiv\equiv$$
$$\mathtt{update}_{\mathtt{i}}^{(2,0)} \circ \pi_1^{(0,0)} \circ (\mathtt{id}_{\mathtt{V_i}}^{(0,0)} \times_{\mathtt{l}} \mathtt{update}_{\mathtt{j}}^{(2,0)}) : \mathtt{V_i} \times \mathtt{V_j} \to \mathbb{1}.$$

These are the archetype properties that we can prove within the scope of the $\mathscr{L}_{st+exc}$. 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 `IMP+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 $\mathscr{L}_{st+exc}$

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

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

$$
\begin{array}{rcc}
\mathtt{daExp}\ \mathtt{n} & \Rightarrow & (\mathtt{constant}\ \mathtt{n})^{(0,0)} \\
\mathtt{daExp}\ \mathtt{x} & \Rightarrow & (\mathtt{lookup_x})^{(1,0)} \\
\mathtt{daExp}\ (\mathtt{a_1} + \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{add})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)} \\
\mathtt{daExp}\ (\mathtt{a_1} \times \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{mlt})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)} \\
\mathtt{daExp}\ (\mathtt{a_1} - \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{subt})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)} \\
\mathtt{dbExp}\ \mathtt{b} & \Rightarrow & (\mathtt{constant}\ \mathtt{b})^{(0,0)} \\
\mathtt{dbExp}\ (\mathtt{a_1} \overset{?}{=} \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{chkeq})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)} \\
\mathtt{dbExp}\ (\mathtt{a_1} \overset{?}{\neq} \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{chkneq})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)} \\
\mathtt{dbExp}\ (\mathtt{a_1} \overset{?}{>} \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{chkgt})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)} \\
\mathtt{dbExp}\ (\mathtt{a_1} \overset{?}{<} \mathtt{a_2}) & \Rightarrow & (\mathtt{tpure}\ \mathtt{chklt})^{(0,0)} \circ \langle \mathtt{daExp}\ \mathtt{a_1}, \mathtt{daExp}\ \mathtt{a_2} \rangle_{\mathtt{l}}^{(\mathtt{d},0)}
\end{array}
$$

$$\texttt{dbExp}\ (\texttt{b}_1 \wedge \texttt{b}_2) \;\Rightarrow\; (\texttt{tpure andB})^{(0,0)} \circ \langle \texttt{dbExp}\ \texttt{b}_1, \texttt{dbExp}\ \texttt{b}_2 \rangle_1^{(\texttt{d},0)}$$

$$\texttt{dbExp}\ (\texttt{b}_1 \vee \texttt{b}_2) \;\Rightarrow\; (\texttt{tpure orB})^{(0,0)} \circ \langle \texttt{dbExp}\ \texttt{b}_1, \texttt{dbExp}\ \texttt{b}_2 \rangle_1^{(\texttt{d},0)}$$

$$\texttt{dbExp}\ (\neg \texttt{b}) \;\Rightarrow\; (\texttt{tpure notB})^{(0,0)} \circ \texttt{dbExp}\ \texttt{b}^{(\texttt{d},0)}$$

In "dbExp b" (6^{th} 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 [(x)].

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., $\langle \texttt{daExp}\ \texttt{a}_1, \texttt{daExp}\ \texttt{a}_2 \rangle_1^{(\texttt{d},0)}$, we use the following strategy:

$$\texttt{d} := \texttt{let}\ \texttt{f}^{(\texttt{d}_1,0)} = \texttt{daExp}(\texttt{a}_1)\ \texttt{in}\ \texttt{let}\ \texttt{g}^{(\texttt{d}_2,0)} = \texttt{daExp}(\texttt{a}_2)\ \texttt{in}\ \texttt{max}(\texttt{d}_1, \texttt{d}_2).$$

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 lpi b f: $\mathbb{1} \to \mathbb{1}$, 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 pbl: $\mathbb{B} \to \mathbb{1}+\mathbb{1}$ forms a bridge between the usual Boolean data type and its correspondence in the decorated settings which is the type $\mathbb{1}+\mathbb{1}$.

$$\texttt{lpi}\ (\texttt{b} : \texttt{term}\ \mathbb{1}\ (\mathbb{1}+\mathbb{1}))\ (\texttt{f} : \texttt{term}\ \mathbb{1}\ \mathbb{1}) \;:=\; \texttt{tpure}\ (\lambda \texttt{x} : \mathbb{1}.\texttt{x}).$$

$$\texttt{pbl} \;:=\; \texttt{tpure}\ (\texttt{bool\_to\_two}).$$

where $\texttt{bool\_to\_two}\ (\texttt{b} : \texttt{bool}) \;:=\; (\texttt{if b then}\ (\texttt{inl void})\ \texttt{else}\ (\texttt{inr void})).$

such that $\texttt{void} : \mathbb{1}$ is the unique constructor of the type $\mathbb{1}$, and

$\texttt{inl},\ \texttt{inr} : \mathbb{1} \to (\mathbb{1}+\mathbb{1})$ are the canonical inclusions.

The rule $(\texttt{imp}_6)$ (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 $(\texttt{imp}_2)$ and $(\texttt{imp}_4)$ by replacing *false* into *true* we get $(\texttt{imp}_3)$ and $(\texttt{imp}_5)$ that are not explicitly stated in Figure 18.

Lemma 7.2. $\texttt{pbl}^{(0,0)} \circ (\texttt{constant}\ \mathit{false})^{(0,0)} \equiv\equiv \texttt{in}_2$.

Proof: unfolding all term definitions, we have $\texttt{tpure}\ (\lambda \texttt{b} : \texttt{bool}.\ \texttt{if b then}\ (\texttt{inl void})\ \texttt{else}\ (\texttt{inr void}))$ $\circ\ \texttt{tpure}\ (\lambda\ \_ : \texttt{void.true}) \equiv\equiv \texttt{tpure inl}$. Now, we obtain $\forall \texttt{x} : \mathbb{1}, \texttt{inl void} = \texttt{inl x}$ by first rewriting tcomp from left to right, and then applying $\texttt{imp}_6$ which is trivial since Leibniz equality '=' is reflexive. □

Lemma 7.3. $\texttt{pbl}^{(0,0)} \circ (\texttt{constant}\ \mathit{true})^{(0,0)} \equiv\equiv \texttt{in}_1$.

(x) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/Functions.v

$$(\mathtt{imp}_1)\ \frac{\forall \mathtt{p},\ \mathtt{q} : \mathbb{Z},\ (\mathtt{f} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z})}{\mathtt{tpure\ f} \circ \langle \mathtt{constant\ p}, \mathtt{constant\ q} \rangle_1 \equiv\equiv (\mathtt{constant\ f(p,q)})}$$

$$(\mathtt{imp}_2)\ \frac{\forall \mathtt{p},\ \mathtt{q} : \mathbb{Z},\ (\mathtt{f} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B})\ \ \mathtt{f(p,q)} = \mathit{false}}{\mathtt{tpure\ f} \circ \langle \mathtt{constant\ p}, \mathtt{constant\ q} \rangle_1 \equiv\equiv \mathtt{constant}\ \mathit{false}}$$

$$(\mathtt{imp}_4)\ \frac{\forall \mathtt{p},\ \mathtt{q} : \mathbb{B},\ (\mathtt{f} : \mathbb{B} \times \mathbb{B} \to \mathbb{B})\ \ \mathtt{f(p,q)} = \mathit{false}}{\mathtt{tpure\ f} \circ \langle \mathtt{constant\ p}, \mathtt{constant\ q} \rangle_1 \equiv\equiv \mathtt{constant}\ \mathit{false}}$$

$$(\mathtt{imp-li})\ \frac{\forall (\mathtt{b} : \mathtt{term}\ \mathbb{1}\ (\mathbb{1} + \mathbb{1}))\ (\mathtt{f} : \mathtt{term}\ \mathbb{1}\ \mathbb{1})}{\mathtt{lpi\ b\ f} \equiv\equiv \left[(\mathtt{lpi\ b\ f}) \circ \mathtt{f} \middle| \mathtt{id} \right]_1 \circ \mathtt{b}}$$

$$(\mathtt{imp}_6)\ \frac{(\forall \mathtt{x},\ \mathtt{f\ x\ =\ g\ x})}{\mathtt{tpure\ f} \equiv \mathtt{tpure\ g}}$$

Figure 18: Additional rules on pure terms: IMP+Exc specific

Proof: It follows the same steps with the proof of Lemma 7.2 □

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 $\mathbb{1} \to \mathbb{1}$, in $\mathtt{throw}_\mathtt{e}^{(0,1)} := [\,]_\mathtt{Y}^{(0,0)} \circ \mathtt{tag}_\mathtt{e}^{(0,1)} : \mathtt{EV}_\mathtt{e} \to \mathtt{Y}$, we replace $\mathtt{EV}_\mathtt{e}$ and Y with $\mathbb{1}$. This means that we stick to a single exceptional value (parameter), for each exception name $\mathtt{e} \in \mathtt{EName}$.

Below, we recursively define the IMP+Exc commands within $\mathscr{L}_{st+exc}$ using a translator function dCmd which establishes a decorated term of type $\mathtt{term}\ \mathbb{1}\ \mathbb{1}$ out of an input command:

$$\begin{array}{rcl}
\mathtt{dCmd\ (SKIP)} & \Rightarrow & (\mathtt{id}\ \mathbb{1})^{(0,0)} \\
\mathtt{dCmd}\ (\mathtt{x} \triangleq \mathtt{a}) & \Rightarrow & (\mathtt{update}_\mathtt{x})^{(2,0)} \circ\ (\mathtt{daExp\ a})^{(\mathtt{d}_1,0)} \\
\mathtt{dCmd}\ (\mathtt{c}_1;\mathtt{c}_2) & \Rightarrow & (\mathtt{dCmd\ c}_2)^{(\mathtt{d}_1,\mathtt{d}_2)} \circ (\mathtt{dCmd\ c}_1)^{(\mathtt{k}_1,\mathtt{k}_2)} \\
\mathtt{dCmd}\ (\mathtt{if\ b\ then\ c}_1\ \mathtt{else\ c}_2) & \Rightarrow & \left[\mathtt{dCmd\ c}_1 \middle| \mathtt{dCmd\ c}_2 \right]_1^{(\mathtt{d}_1,\mathtt{d}_2)} \circ\ \mathtt{pbl}^{(0,0)} \circ (\mathtt{dbExp\ b})^{(\mathtt{d}_3,0)} \\
\mathtt{dCmd\ (while\ b\ do\ c)} & \Rightarrow & \left[(\mathtt{lpi\ (pbl} \circ (\mathtt{dbExp\ b}))\ (\mathtt{dCmd\ c})) \circ (\mathtt{dCmd\ c}) \middle| \mathtt{id} \right]_1^{(\mathtt{d}_1,\mathtt{d}_2)} \\
 & & \circ\ \mathtt{pbl}^{(0,0)} \circ (\mathtt{dbExp\ b})^{(\mathtt{d}_3,0)} \\
\mathtt{dCmd\ (THROW\ e)} & \Rightarrow & \mathtt{throw\ e}^{(0,1)} \\
\mathtt{dCmd}\ (\mathtt{TRY\ c}_1\ \mathtt{CATCH\ e} \Rightarrow \mathtt{c}_2) & \Rightarrow & \mathtt{try\ (dCmd\ c}_1)\ \mathtt{catch}\ (\mathtt{e} \Rightarrow (\mathtt{dCmd\ c}_2))^{(\mathtt{d}_1,\mathtt{d}_2)}
\end{array}$$

Remark 7.5. To calculate the decorations $\mathtt{d}_1$ and $\mathtt{d}_2$ (or $\mathtt{k}_1$ and $\mathtt{k}_2$), we use the following strategy:

$$\mathtt{d}_1 := \mathtt{let\ f}^{(\mathtt{d}'_1,\mathtt{d}'_2)} = \mathtt{dCmd(c_1)\ in\ let\ g}^{(\mathtt{d}'_3,\mathtt{d}'_4)} = \mathtt{dCmd(c_2)\ in\ max(d'_1,d'_3)}.$$
$$\mathtt{d}_2 := \mathtt{let\ f}^{(\mathtt{d}'_1,\mathtt{d}'_2)} = \mathtt{dCmd(c_1)\ in\ let\ g}^{(\mathtt{d}'_3,\mathtt{d}'_4)} = \mathtt{dCmd(c_2)\ in\ max(d'_2,d'_4)}.$$

(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, `dCmd` function outputs terms at most wih decoration (1) with respect to the exception effect.

In Figure 19, the diagram on the left schematizes the command `if b then` c_1 `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 know 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 `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 $\mathtt{id}_{\mathbb{1}}$ in execution forcing the loop to terminate. Recall that the case distinction in the diagrams are provided by the term inclusions.

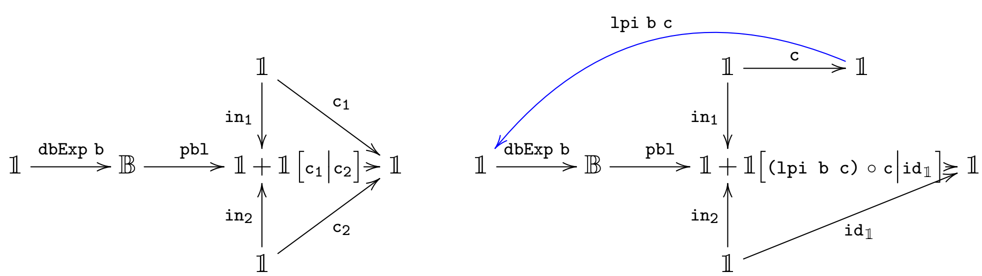


Figure 19: (`cond b` c_1 c_2) and (`while b do c`) in $\mathscr{L}_{st+exc}$

Figure 20 respectively visualizes the formal behaviors of `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 $\mathbb{1}$.

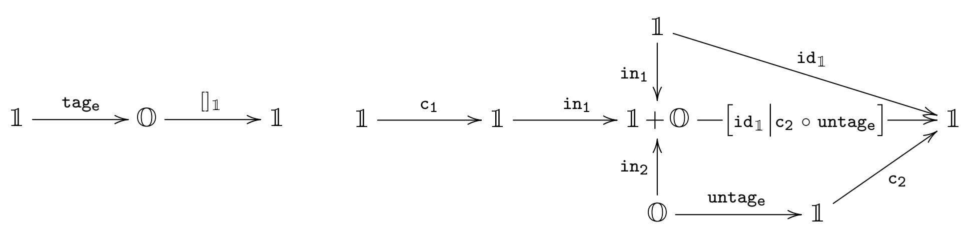


Figure 20: (`THROW e`) and (`TRY` c_1 `CATCH e` $\Rightarrow$ c_2) in $\mathscr{L}_{st+exc}$

We now encode the `IMP+Exc` denotational semantics, with the $\mathscr{L}_{st+exc}$ as the target language, in Coq. Arithmetic and Boolean expressions are inductively forming new Coq `Types`, called `aExp` and `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 `bconst`.

```
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 chkge 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
   | THROW     : EName → Cmd
   | TRY_CATCH : EName → Cmd → Cmd → Cmd.
```

We now interpret the dCmd function in Coq using the below fixpoint:

```
Fixpoint dCmd (c: Cmd): (term unit unit) ≜
 match c with
   | skip              ⇒ (@id unit)
   | sequence c0 c1    ⇒ (dCmd c1) o (dCmd c0)
   | assign j e0       ⇒ (update j) o (daExp e0)
   | cond b c2 c3      ⇒ copair (dCmd c2) (dCmd c3) o (pbl o (dbExp b))
   | while b c4        ⇒ (copair (lpi (pbl o (dbExp b)) (dCmd c4) o (dCmd c4)) (@id unit)) o
                         (pbl o (dbExp b))
   | THROW e           ⇒ (throw unit e)
   | TRY_CATCH e c1 c2 ⇒ (try_catch e (dCmd c1) (dCmd c2))
 end.
```

Now, we retain sufficient material to state and prove equivalences between programs written in IMP+Exc. Also, the discussed Coq implementation allows us to certfy them in Coq.

7.1 Program equivalence proofs

In this section, we finally prove equivalences of several programs written in IMP+Exc, using the denotational semantics characterized within the scope of the logic $\mathscr{L}_{st+exc}$. Note that for the sake of simplicity, we will use $\mathtt{u_x}$, $\mathtt{l_x}$, $(\mathtt{t\ op})$ and $(\mathtt{c\ p})$ instead of $(\mathtt{update\ x})^{(2,0)}$, $(\mathtt{lookup\ x})^{(1,0)}$, $(\mathtt{tpure\ op})^{(0,0)}$ and $(\mathtt{constant\ p})^{(0,0)}$, respectively.

Remark 7.6. Recall that the use of term products is to impose some order of term evaluation on the mutable state. 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 through out 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 $\forall \mathtt{x} \in \mathtt{Loc}\ \mathtt{p},\mathtt{q} : \mathbb{Z},\ \mathtt{u_x} \circ (\mathtt{c\ p}) \circ \mathtt{u_x} \circ (\mathtt{c\ q}) \equiv\equiv \mathtt{u_x} \circ (\mathtt{c\ p})$
2. commutation update-update $\forall \mathtt{x} \neq \mathtt{y} \in \mathtt{Loc}\ \mathtt{p},\mathtt{q} : \mathbb{Z},\ \mathtt{u_x} \circ (\mathtt{c\ p}) \circ \mathtt{u_y} \circ (\mathtt{c\ q}) \equiv\equiv \mathtt{u_y} \circ (\mathtt{c\ q}) \circ \mathtt{u_x} \circ (\mathtt{c\ p})$
3. commutation-lookup-constant-update $\forall \mathtt{x} \in \mathtt{Loc}, \mathtt{p},\mathtt{q} \in \mathbb{Z},\ \langle \mathtt{l_x}, (\mathtt{c\ q}) \rangle_1 \circ \mathtt{u_x} \circ (\mathtt{c\ p}) \equiv\equiv \langle (\mathtt{c\ p}), (\mathtt{c\ q}) \rangle_1 \circ \mathtt{u_x} \circ (\mathtt{c\ p})$

Figure 21: Primitive properties of the state: IMP+Exc specific

Remark 7.7. Below, we state three lemmata using the 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 through out 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* $\mathtt{f}, \mathtt{g}$ $(\mathtt{doesNotThrowTC}\,(\mathtt{f}) = \mathtt{true}, \mathtt{doesNotThrowTC}\,(\mathtt{g}) = \mathtt{true})$ *and* $\mathtt{b} \in \{\mathtt{true}, \mathtt{false}\}$*, if program pieces* `prog1` *and* `prog2` *are given as in the following listings, then* $\mathtt{dCmd}\,(\mathtt{prog1}) \equiv\equiv \mathtt{dCmd}\,(\mathtt{prog2})$.

Listing 1: prog1

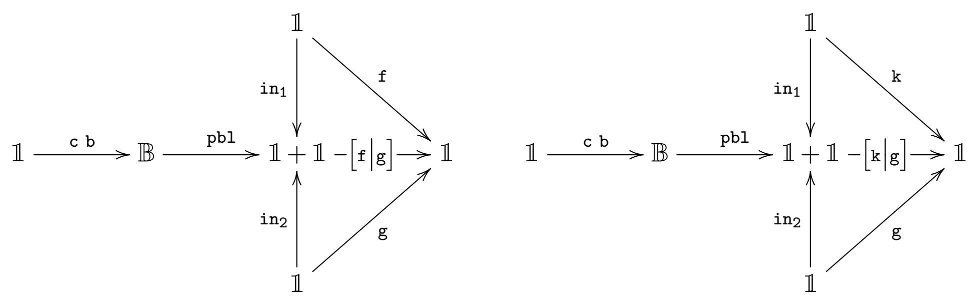
```
/* prog1 */
if b then f else g;
```

Listing 2: prog2

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

Note that the function $\mathtt{doesNotThrowTC} : \mathtt{cmd} \to \mathtt{Bool}$ takes any command, recursively checks whether the input involves either `THROW` or `TRY/CATCH`, and returns `true` if that is the case; `false` otherwise. Browse this function, in a Coq implementation, here [(xiv)].

Proof: We sketch the diagrams of both programs below:



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

$$\left[\mathtt{f}\middle|\mathtt{g}\right]_1 \circ \mathtt{pbl} \circ \mathtt{c\;b} \equiv\equiv \left[\mathtt{k}\middle|\mathtt{g}\right]_1 \circ \mathtt{pbl} \circ \mathtt{c\;b}. \tag{1}$$

Using the rules of the logic $\mathscr{L}_{st+exc}$, 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 = false`, by unfolding the definitions of `pbl` and $(\mathtt{c}\;\mathit{false})$, we have

$$\left[\mathtt{f}\middle|\mathtt{g}\right]_1 \circ \mathtt{t}\;(\mathtt{bool\_to\_two}) \circ \mathtt{t}\;(\lambda \mathtt{x} : \mathtt{unit.false}) \equiv\equiv \left[\mathtt{k}\middle|\mathtt{g}\right]_1 \circ \mathtt{t}\;(\mathtt{bool\_to\_two}) \circ \mathtt{t}\;(\lambda \mathtt{x} : \mathtt{unit.false}). \tag{2}$$

We rewrite (tcomp) on both sides, and get

$$\left[\mathtt{f}\middle|\mathtt{g}\right]_1 \circ \mathtt{t}\;(\lambda \mathtt{x} : \mathtt{unit.bool\_to\_two\;false}) \equiv\equiv \left[\mathtt{k}\middle|\mathtt{g}\right]_1 \circ \mathtt{t}\;(\lambda \mathtt{x} : \mathtt{unit.bool\_two\;false}). \tag{3}$$

(xiv) https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/IMPEX_to_COQ.v#L148

Now, we cut

$$\mathtt{t}\ (\lambda \mathtt{x} : \mathtt{unit}.\mathtt{bool\_to\_two}\ \mathtt{false}) \equiv\equiv \mathtt{in}_2 \quad (4)$$

and rewrite it back in the goal. So that we obtain

$$[\mathtt{f}|\mathtt{g}]_1 \circ \mathtt{in}_2 \equiv\equiv [\mathtt{k}|\mathtt{g}]_1 \circ \mathtt{in}_2. \quad (5)$$

Then, we use (s_lcopair_eq), and finally have $\mathtt{g} \equiv\equiv \mathtt{g}$ which is trivial since $\equiv\equiv$ is reflexive. It remains to show that the cut statement in Equation 4 holds. By simplifying $\mathtt{t}\ (\lambda \mathtt{x} : \mathtt{unit}.\ \mathtt{bool\_to\_two}\ \mathtt{false})$ and unfolding $\mathtt{in}_2$, we have

$$\mathtt{t}\ (\lambda \mathtt{x} : \mathtt{unit}.\mathtt{inr}\ \mathtt{x}) \equiv\equiv \mathtt{t}\ (\mathtt{inr}). \quad (6)$$

Now, we apply (imp_6) and get

$$\forall \mathtt{x} : \mathtt{unit}, \mathtt{inr}\ \mathtt{x} = \mathtt{inr}\ \mathtt{x} \quad (7)$$

which is trivial since the Leibniz equality '$=$' is reflexive.

If `b = true`, by following above procedure with `true` (instead of `false`) we first handle

$$[\mathtt{f}|\mathtt{g}]_1 \circ \mathtt{in}_1 \equiv\equiv [\mathtt{k}|\mathtt{g}]_1 \circ \mathtt{in}_1 \quad (8)$$

and then freely convert $\equiv\equiv$ into $\equiv\sim$. There, rewriting the rule (w_lcopair_eq) yields $\mathtt{f} \equiv\sim \mathtt{k}$. We unfold `k` with $\mathtt{b} = \mathtt{true}$ and get

$$\mathtt{f} \equiv\sim [\mathtt{f}|\mathtt{g}]_1 \circ \mathtt{in}_1. \quad (9)$$

Now by rewriting (w_lcopair_eq), we have $\mathtt{f} \equiv\sim \mathtt{f}$. This is again trivial, since the equality sort $\equiv\sim$ is reflexive. □

Lemma 7.10. *For all* $\mathtt{x} : \mathtt{Loc}$*, if program pieces* `prog3` *and* `prog4` *are given as in the following listings, then* $\mathtt{dCmd}\ (\mathtt{prog3}) \equiv\equiv \mathtt{dCmd}\ (\mathtt{prog4})$.

Listing 3: prog3

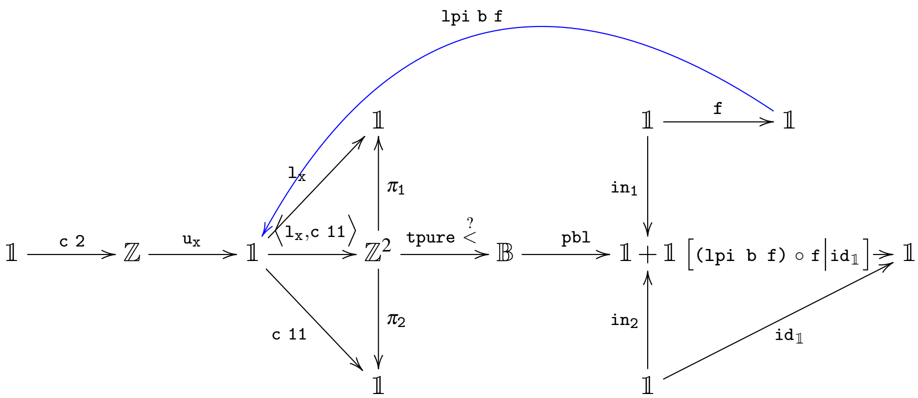
```
/* prog3 */
x ≜ 2;
while (x <? 11)
   do (x ≜ x + 4);
```

Listing 4: prog4

```
/* prog4 */
x ≜ 14;
```

Proof: In the proof structure we intend to reduce `prog3`, first dealing with the pre-loop assignments and the looping pre-condition. Since it evaluates into *true*, in the second step we identify things related to the first loop iteration. The third step primarily studies the second and then the third loop iteration after which the looping pre-condition switches to *false*. Finally, we explain the program termination and show that `prog3` does exactly the same state manipulation with `prog4`. Note also that we do not need to check the results they returned, since all `IMP+Exc` commands, thus programs, return `void: U`.

Below is the sketch of prog3:



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

$$\mathbb{1} \xrightarrow{\mathtt{c}\ 14} \mathbb{Z} \xrightarrow{\mathtt{u_x}} \mathbb{1}$$

which is actually prog4 when sketched.

1. Initially, we have

$$\left[(\mathtt{lpi\ b\ f})\circ \mathtt{f}\middle|\mathtt{id}_{\mathbb{1}}\right]\circ \mathtt{pbl}\circ(\mathtt{t}\ \overset{?}{<})\circ\left\langle \mathtt{l_x},(\mathtt{c}\ 11)\right\rangle\circ \mathtt{u_x}\circ(\mathtt{c}\ 2)\equiv\equiv \mathtt{u_x}\circ(\mathtt{c}\ 14). \tag{10}$$

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

$$\left[(\mathtt{lpi\ b\ f})\circ \mathtt{f}\middle|\mathtt{id}_{\mathbb{1}}\right]\circ \mathtt{pbl}\circ(\mathtt{t}\ \overset{?}{<})\circ\left\langle (\mathtt{c}\ 2),(\mathtt{c}\ 11)\right\rangle\circ \mathtt{u_x}\circ(\mathtt{c}\ 2)\equiv\equiv \mathtt{u_x}\circ(\mathtt{c}\ 14). \tag{11}$$

Since the looping pre-condition $(\mathtt{t}\ \overset{?}{<})\circ\left\langle(\mathtt{c}\ 2),(\mathtt{c}\ 11)\right\rangle$ evaluates into $(\mathtt{c}\ \mathit{true})$, and due to ($imp_3$), we have

$$\left[(\mathtt{lpi\ b\ f})\circ \mathtt{f}\middle|\mathtt{id}_{\mathbb{1}}\right]\circ \mathtt{pbl}\circ(\mathtt{c}\ \mathit{true})\circ \mathtt{u_x}\circ(\mathtt{c}\ 2)\equiv\equiv \mathtt{u_x}\circ(\mathtt{c}\ 14). \tag{12}$$

By rewriting the Lemma 7.3, we get

$$\left[(\mathtt{lpi\ b\ f})\circ \mathtt{f}\middle|\mathtt{id}_{\mathbb{1}}\right]\circ \mathtt{in}_1\circ \mathtt{u_x}\circ(\mathtt{c}\ 2)\equiv\equiv \mathtt{u_x}\circ(\mathtt{c}\ 14). \tag{13}$$

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

$$(\mathtt{lpi\ b\ f})\circ \mathtt{f}\circ \mathtt{u_x}\circ(\mathtt{c}\ 2)\equiv\sim \mathtt{u_x}\circ(\mathtt{c}\ 14) \tag{14}$$

in which the second appearance of f unfolds into

$$(\mathtt{lpi\ b\ f})\circ \mathtt{u_x}\circ(\mathtt{t}\ +)\circ\left\langle \mathtt{l_x},\mathtt{c}\ 4\right\rangle\circ \mathtt{u_x}\circ(\mathtt{c}\ 2)\equiv\sim \mathtt{u_x}\circ(\mathtt{c}\ 14). \tag{15}$$

Since, there is no exceptional case, we are freely back to $\equiv\equiv$. By rewriting `commutation −lookup −constant −update`, we obtain

$$(\texttt{lpi b f}) \circ \mathtt{u_x} \circ (\texttt{t} +) \circ \langle \texttt{c } 2, \texttt{c } 4 \rangle \circ \mathtt{u_x} \circ (\texttt{c } 2) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{16}$$

The rule (imp_1) gives

$$(\texttt{lpi b f}) \circ \mathtt{u_x} \circ (\texttt{c } 6) \circ \mathtt{u_x} \circ (\texttt{c } 2) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{17}$$

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

$$(\texttt{lpi b f}) \circ \mathtt{u_x} \circ (\texttt{c } 6) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{18}$$

2. For the second loop iteration, rewriting (imp-li) gives

$$\big[(\texttt{lpi b f}) \circ \texttt{f} \big| \mathtt{id_{\mathbb{1}}}\big] \circ \texttt{pbl} \circ (\texttt{t} \overset{?}{<}) \circ \big\langle \mathtt{l_x}, (\texttt{c } 11) \big\rangle \circ \mathtt{u_x} \circ (\texttt{c } 6) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{19}$$

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

$$(\texttt{lpi b f}) \circ \mathtt{u_x} \circ (\texttt{c } 10) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{20}$$

3. In the third iteration, rewriting the (imp-li) gives

$$\big[(\texttt{lpi b f}) \circ \texttt{f} \big| \mathtt{id_{\mathbb{1}}}\big] \circ \texttt{pbl} \circ (\texttt{t} \overset{?}{<}) \circ \big\langle \mathtt{l_x}, (\texttt{c } 11) \big\rangle \circ \mathtt{u_x} \circ (\texttt{c } 10) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{21}$$

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

$$(\texttt{lpi b f}) \circ \mathtt{u_x} \circ (\texttt{c } 14) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{22}$$

4. In the fourth step, rewriting the (imp-li) gives

$$\big[(\texttt{lpi b f}) \circ \texttt{f} \big| \mathtt{id_{\mathbb{1}}}\big] \circ \texttt{pbl} \circ (\texttt{t} \overset{?}{<}) \circ \big\langle \mathtt{l_x}, (\texttt{c } 11) \big\rangle \circ \mathtt{u_x} \circ (\texttt{c } 14) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{23}$$

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

$$\big[(\texttt{lpi b f}) \circ \texttt{f} \big| \mathtt{id_{\mathbb{1}}}\big] \circ \texttt{pbl} \circ (\texttt{t} \overset{?}{<}) \circ \big\langle (\texttt{c } 14), (\texttt{c } 11) \big\rangle \circ \mathtt{u_x} \circ (\texttt{c } 14) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{24}$$

Finally here, the looping pre-condition $(\texttt{t} \overset{?}{<}) \circ \big\langle (\texttt{c } 14), (\texttt{c } 11) \big\rangle$ evaluates into (`c` *false*) yielding

$$\big[(\texttt{lpi b f}) \circ \texttt{f} \big| \mathtt{id_{\mathbb{1}}}\big] \circ \texttt{pbl} \circ (\texttt{c } \mathit{false}) \circ \mathtt{u_x} \circ (\texttt{c } 14) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{25}$$

We rewrite the Lemma 7.2, and get

$$\big[(\texttt{lpi b f}) \circ \texttt{f} \big| \mathtt{id_{\mathbb{1}}}\big] \circ \mathtt{in_2} \circ \mathtt{u_x} \circ (\texttt{c } 14) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14). \tag{26}$$

Now we rewrite (s_lcopair_eq) and handle

$$\mathtt{id_{\mathbb{1}}} \circ \mathtt{u_x} \circ (\texttt{c } 14) \equiv\equiv \mathtt{u_x} \circ (\texttt{c } 14) \tag{27}$$

which is trivial, since the identity term disappears when to compose and the equality sort $\equiv\equiv$ is reflexive. □

Lemma 7.11. *For each* x y : Loc*,* e : EName*, if program pieces* prog5 *and* prog6 *are given as in the following listings, then* dCmd (prog5) ≡≡ dCmd (prog6).

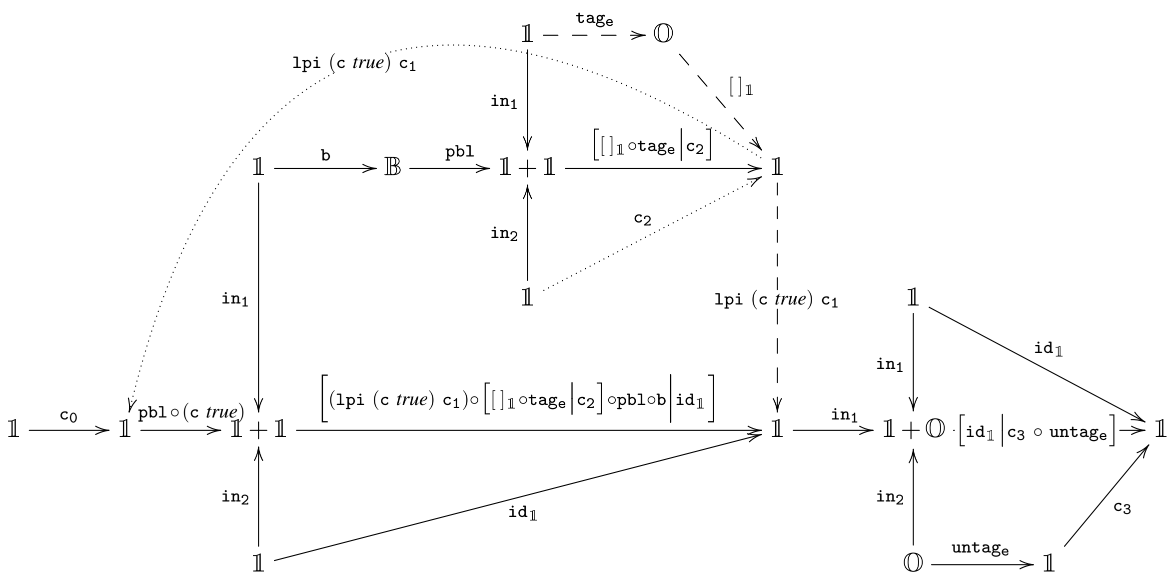
Listing 5: prog5

```
/* 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: prog6

```
/* 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 prog5:



where $\mathtt{b} = (\mathtt{x} \overset{?}{\leq} 0)$, $\mathtt{c}_0 = (\mathtt{x} \triangleq 1; \mathtt{y} \triangleq 20)$, $\mathtt{c}_1 = (\mathtt{if}\ (\mathtt{x} \overset{?}{\leq} 0)\ \mathtt{then}\ (\mathtt{THROW}\ \mathtt{e})\ \mathtt{else}\ (\mathtt{x} \triangleq \mathtt{x} - 1))$, $\mathtt{c}_2 = (\mathtt{x} \triangleq \mathtt{x} - 1)$, $\mathtt{c}_3 = (\mathtt{y} \triangleq 7)$. 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 $\mathscr{L}_{st+exc}$, we can reduce the above diagram into the one given below with respect to the equality sort ≡≡:

$$\mathbb{1} \xrightarrow{\mathtt{c}\ 0} \mathbb{Z} \xrightarrow{\mathtt{u_x}} \mathbb{1} \xrightarrow{\mathtt{c}\ 7} \mathbb{Z} \xrightarrow{\mathtt{u_y}} \mathbb{1}$$

which is actually the `prog6` when sketched.

1. Initially, we have

$$\downarrow\Big(\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ\Big[(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]\circ\mathtt{pbl}\circ\mathtt{b}\Big|\mathtt{id}_{\mathbb{1}}\Big]\circ\mathtt{pbl}\circ(\mathtt{c}\ \mathit{true})\Big)$$
$$\circ\,\mathtt{u_y}\circ(\mathtt{c}\ 20)\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\equiv\equiv\mathtt{u_y}\circ(\mathtt{c}\ 7)\circ\mathtt{u_x}\circ(\mathtt{c}\ 0). \quad (28)$$

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

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ\Big[(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]\circ\mathtt{pbl}\circ\mathtt{b}\Big|\mathtt{id}_{\mathbb{1}}\Big]$$
$$\circ\,\mathtt{pbl}\circ(\mathtt{c}\ \mathit{true})\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\equiv\sim\mathtt{u_y}\circ(\mathtt{c}\ 7)\circ\mathtt{u_x}\circ(\mathtt{c}\ 0). \quad (29)$$

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

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ\Big[(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]\circ\mathtt{pbl}\circ\mathtt{b}\Big|\mathtt{id}_{\mathbb{1}}\Big]$$
$$\circ\,\mathtt{pbl}\circ(\mathtt{c}\ \mathit{true})\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u_x}\circ(\mathtt{c}\ 0)\circ\mathtt{u_y}\circ(\mathtt{c}\ 7). \quad (30)$$

Rewriting Lemma 7.3 yields

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ\Big[(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]\circ\mathtt{pbl}\circ\mathtt{b}\Big|\mathtt{id}_{\mathbb{1}}\Big]$$
$$\circ\,\mathtt{in}_1\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u_x}\circ(\mathtt{c}\ 0)\circ\mathtt{u_y}\circ(\mathtt{c}\ 7). \quad (31)$$

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

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]$$
$$\circ\,\mathtt{pbl}\circ\mathtt{b}\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u_x}\circ(\mathtt{c}\ 0). \quad (32)$$

By unfolding `b`, we have

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]$$
$$\circ\,\mathtt{pbl}\circ(\mathtt{t}\ \overset{?}{\leq})\circ\big\langle\mathtt{l_x}\ (\mathtt{c}\ 0)\big\rangle\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u_x}\circ(\mathtt{c}\ 0)\circ\mathtt{u_y}\circ(\mathtt{c}\ 7). \quad (33)$$

By rewriting the lemma `commutation− lookup− constant− update`, we obtain

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]$$
$$\circ\,\mathtt{pbl}\circ(\mathtt{t}\ \overset{?}{\leq})\circ\big\langle(\mathtt{c}\ 1),(\mathtt{c}\ 0)\big\rangle\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u_x}\circ(\mathtt{c}\ 0)\circ\mathtt{u_y}\circ(\mathtt{c}\ 7). \quad (34)$$

We rewrite the rule (imp_2), and get

$$\big[\mathtt{id}_{\mathbb{1}}\big|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\big]\circ\mathtt{in}_1\circ(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)$$
$$\circ\,\big[[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\big|\mathtt{c}_2\big]\circ\mathtt{pbl}\circ(\mathtt{c}\ \mathit{false})\circ\mathtt{u_x}\circ(\mathtt{c}\ 1)\circ\mathtt{u_y}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u_x}\circ(\mathtt{c}\ 0)\circ\mathtt{u_y}.\circ(\mathtt{c}\ 7). \quad (35)$$

Rewriting the Lemma 7.2 yields

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \left[[\,]_{\mathbb{1}} \circ \mathtt{tag}_{\mathtt{e}} \middle| \mathtt{c}_2\right] \circ \mathtt{in}_2 \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 1) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}}. \circ (\mathtt{c}\ 7). \quad (36)$$

We now rewrite (s_lcopair_eq) which gives

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \mathtt{c}_2 \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 1) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 7). \quad (37)$$

Here, by unfolding $\mathtt{c}_2$, we have

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{t}\ -) \circ \langle \mathtt{l}_{\mathtt{x}}, (\mathtt{c}\ 1)\rangle \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 1) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 7). \quad (38)$$

Rewriting the lemma $\mathtt{commutation} - \mathtt{lookup} - \mathtt{constant} - \mathtt{update}$ gives

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{t}\ -) \circ \langle (\mathtt{c}\ 1), (\mathtt{c}\ 1)\rangle \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 1) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ 0(\mathtt{c}\ 7). \quad (39)$$

We rewrite (imp_1), and get

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 1) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 7). \quad (40)$$

We again rewrite the lemma `commutation-update-update`, and obtain

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 7). \quad (41)$$

3. We re-iterate the loop via (imp-li), and have

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ \left[(\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \mathtt{c}_1 \middle| \mathtt{id}\right] \circ \mathtt{pbl} \circ (\mathtt{c}\ \mathit{true}) \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 7). \quad (42)$$

We rewrite Lemma 7.3, (w_lcopair_eq), then unfold $\mathtt{c}_1$, and get:

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \left[\mathtt{throw}\ \mathtt{e}\ \mathbb{1} \middle| \mathtt{c}_2\right] \circ \mathtt{pbl} \circ (\mathtt{t}\ \overset{?}{\leq}) \circ \langle \mathtt{l}_{\mathtt{x}}, (\mathtt{c}\ 0)\rangle \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20). \quad (43)$$

By rewriting $\mathtt{commutation} - \mathtt{lookup} - \mathtt{constant} - \mathtt{update}$, ($\mathrm{imp}_3$) and Lemma 7.3, we have

$$\left[\mathtt{id}_{\mathbb{1}} \middle| \mathtt{c}_3 \circ \mathtt{untag}_{\mathtt{e}}\right] \circ \mathtt{in}_1 \circ (\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1) \circ \left[\mathtt{throw}\ \mathtt{e}\ \mathbb{1} \middle| \mathtt{c}_2\right] \circ \mathtt{in}_1 \circ \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20) \equiv\sim \mathtt{u}_{\mathtt{x}} \circ (\mathtt{c}\ 0) \circ \mathtt{u}_{\mathtt{y}} \circ (\mathtt{c}\ 20). \quad (44)$$

By (w_lcopair_eq), the exception is raised:

$$\left[\mathtt{id}_{\mathbb{1}}\middle|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\right]\circ\mathtt{in}_1\circ\big((\mathtt{lpi}\ (\mathtt{c}\ \mathit{true})\ \mathtt{c}_1)\circ\mathtt{throw\ e}\ \mathbb{1}\big)$$
$$\circ\,\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20). \quad (45)$$

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

$$\left[\mathtt{id}_{\mathbb{1}}\middle|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\right]\circ\mathtt{in}_1\circ[\,]_{\mathbb{1}}\circ\mathtt{tag}_{\mathtt{e}}\circ\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20). \quad (46)$$

4. Here, we first cut $\mathtt{in}_1\circ[\,]_{\mathbb{1}}\equiv\equiv\mathtt{in}_2$, and rewrite it back in the equation. Thus, we have

$$\left[\mathtt{id}_{\mathbb{1}}\middle|\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\right]\circ\mathtt{in}_2\circ\mathtt{tag}_{\mathtt{e}}\circ\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7). \quad (47)$$

By rewriting (s_lcopair_eq), we obtain

$$\mathtt{c}_3\circ\mathtt{untag}_{\mathtt{e}}\circ\mathtt{tag}_{\mathtt{e}}\circ\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7). \quad (48)$$

Since $\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)$ is pure with respect to the exception, we rewrite (eax_1), and get

$$\mathtt{c}_3\circ\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7). \quad (49)$$

Unfolding the definition of the command $\mathtt{c}_3=(\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7))$, we have

$$\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7)\circ\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7). \quad (50)$$

We now rewrite commutation− update− update on the left, and handle

$$\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 20)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7). \quad (51)$$

Finally, it suffices to rewrite interaction− update−update,

$$\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7)\equiv\sim\mathtt{u}_{\mathtt{x}}\circ(\mathtt{c}\ 0)\circ\mathtt{u}_{\mathtt{y}}\circ(\mathtt{c}\ 7). \quad (52)$$

which is trivial since the equality symbol $\equiv\sim$ is reflexive. However, it still remains to prove the previous cut $\mathtt{in}_1\circ[\,]_{\mathbb{1}}\equiv\equiv\mathtt{in}_2$: since everything is pure with respect to the exception, we have

$$\mathtt{in}_1\circ[\,]_{\mathbb{1}}\equiv\sim\mathtt{in}_2. \quad (53)$$

Now, rewriting the rule (w_empty) gives $[\,]_{\mathbb{1}+\mathbb{1}}\equiv\sim[\,]_{\mathbb{1}+\mathbb{1}}$. This is trivial since the equality sort $\equiv\sim$ is reflexive. □

The full Coq proofs of above lemmata can be found here [xv], and the entire implementation there [xvi].

[xv] https://github.com/ekiciburak/impex-on-decorated-logic/blob/master/IMPEX_Proofs.v
[xvi] https://github.com/ekiciburak/impex-on-decorated-logic

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 $\mathscr{L}_{st+exc}$

With the logic $\mathscr{L}_{st+exc}$, no generic program properties such as

$$\mathtt{dCmd}(\mathtt{p}_1) \equiv\equiv \mathtt{dCmd}(\mathtt{p}_2) \implies \forall \mathtt{s}\,\mathtt{s}', \mathtt{eval}\,\mathtt{p}_1\,\mathtt{s}\,\mathtt{s}' \implies \mathtt{eval}\,\mathtt{p}_2\,\mathtt{s}\,\mathtt{s}'$$

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+Exc` language using the logic $\mathscr{L}_{st+exc}$ 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 $\mathtt{L}_0$ and $\mathtt{L}$ such that $\mathtt{L}_0 \subseteq \mathtt{L}$ ($\mathtt{L}_0$ is a sub-logic of $\mathtt{L}$) and a theory $\mathtt{T}$ of $\mathtt{L}$. $\mathtt{T}$ is relatively Hilbert-Post complete with respect to $\mathtt{L}_0$ if (1) at least one sentence is unprovable in $\mathtt{T}$ (not the maximal theory ensuring consistency), and (2) every theory containing $\mathtt{T}$ can be generated from $\mathtt{T}$ and some sentences from $\mathtt{L}_0$. Here, $\mathtt{L}_0$ can be seen as the pure logic that governs the denotational semantics of the effect-free subset of the `IMP` language where $\mathtt{L}$ 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 $\mathscr{L}_{st}$ 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 $\mathscr{L}_{st}$. 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 $\mathscr{L}_{st+exc}$ 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 `IMP+Exc` language over the combined decorated logic $\mathscr{L}_{st+exc}$. This way, we managed prove some strong equalities between `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

A. Bauer and M. Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, 10(4), 2014. doi: 10.2168/LMCS-10(4:9)2014. URL http://dx.doi.org/10.2168/LMCS-10(4:9)2014.

A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015. doi: 10.1016/j.jlamp.2014.02.001. URL http://dx.doi.org/10.1016/j.jlamp.2014.02.001.

N. Benton and A. Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, July 2001. ISSN 0956-7968. doi: 10.1017/S0956796801004099. URL http://dx.doi.org/10.1017/S0956796801004099.

E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500581. URL http://doi.acm.org/10.1145/2500365.2500581.

S. Brookes and K. Van Stone. Monads and Comonads in Intensional Semantics. Technical Report CMU-CS-93-140, Pittsburgh, PA, USA, 1993. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.5695.

Ł. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018. doi: 10.1007/s10817-018-9458-4. URL https://doi.org/10.1007/s10817-018-9458-4.

C. Domínguez and D. Duval. Diagrammatic logic applied to a parameterisation process. *Mathematical Structures in Computer Science*, 20(4):639–654, 2010. doi: 10.1017/S0960129510000150. URL http://dx.doi.org/10.1017/S0960129510000150.

J. Dumas, D. Duval, L. Fousse, and J. Reynaud. A duality between exceptions and states. *Mathematical Structures in Computer Science*, 22(4):719–722, 2012. doi: 10.1017/S0960129511000752. URL http://dx.doi.org/10.1017/S0960129511000752.

J. Dumas, D. Duval, B. Ekici, and D. Pous. Formal verification in Coq of program properties involving the global state effect. In C. Tasson, editor, *25e Journées Francophones des Langages Applicatifs, Fréjus*, Jan. 2014a. URL http://hal.archives-ouvertes.fr/hal-00869230.

J. Dumas, D. Duval, B. Ekici, and J. Reynaud. Certified proofs in programs involving exceptions. In *Joint Proceedings of the MathUI, OpenMath and ThEdu Workshops and Work in Progress track at CICM co-located with Conferences on Intelligent Computer Mathematics (CICM 2014), Coimbra, Portugal, July 7-11, 2014.*, 2014b. URL http://ceur-ws.org/Vol-1186/paper-20.pdf.

J. Dumas, D. Duval, and J. Reynaud. Breaking a monad-comonad symmetry between computational effects. *CoRR*, abs/1402.1051, 2014c. URL http://arxiv.org/abs/1402.1051.

J. Dumas, D. Duval, B. Ekici, D. Pous, and J. Reynaud. Relative hilbert-post completeness for exceptions. In *Mathematical Aspects of Computer and Information Sciences - 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers*, pages 596–610, 2015. doi: 10.1007/978-3-319-32859-1_51. URL http://dx.doi.org/10.1007/978-3-319-32859-1_51.

J. Egger, R. E. Møgelberg, and A. Simpson. The enriched effect calculus: syntax and semantics. *J. Log. Comput.*, 24(3):615–654, 2014. doi: 10.1093/logcom/exs025. URL http://dx.doi.org/10.1093/logcom/exs025.

B. Ekici. *Certification de programmes avec des effets calculatoires*. PhD thesis, 2015. URL https://tel.archives-ouvertes.fr/tel-01250842/document.

A. Filinski. *Controlling Effects*. PhD thesis, 1996.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.

M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3): 70–99, 2006. doi: 10.1016/j.tcs.2006.03.013. URL http://dx.doi.org/10.1016/j.tcs.2006.03.013.

M. Hyland, P. B. Levy, G. D. Plotkin, and J. Power. Combining algebraic effects with continuations. *Theor. Comput. Sci.*, 375(1-3):20–40, 2007. doi: 10.1016/j.tcs.2006.12.026. URL http://dx.doi.org/10.1016/j.tcs.2006.12.026.

B. Jacobs. A formalisation of java's exception mechanism. In *ESOP'01*, pages 284–301, 2001.

B. Jacobs and J. Rutten. An introduction to (co)algebras and (co)induction. In *In: D. Sangiorgi and J. Rutten (eds), Advanced topics in bisimulation and coinduction*, pages 38–99, 2011.

M. Jaskelioff. Modular monad transformers. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 64–79, 2009. doi: 10.1007/978-3-642-00590-9_6. URL http://dx.doi.org/10.1007/978-3-642-00590-9_6.

O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158, 2013. doi: 10.1145/2500365.2500590. URL http://doi.acm.org/10.1145/2500365.2500590.

F. W. Lawvere. *Functorial Semantic of Algebraic Theories (Available with commentary as TAC Reprint 5.)*. PhD thesis, 1963. URL http://matija.pretnar.info/pdf/the-logic-and-handling-of-algebraic-effects.pdf.

P. B. Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, pages 228–242, 1999. doi: 10.1007/3-540-48959-2_17. URL http://dx.doi.org/10.1007/3-540-48959-2_17.

F. Linton. Relative functorial semantics: adjointness results. In *Lecture notes in mathematics*, volume 99, 1969.

F. E. J. Linton. Some aspects of equational theories. In *Proc. Conf. on Categorical Algebra*, pages 84–95, La Jolla, 1966. Springer-Verlag.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73564. URL http://doi.acm.org/10.1145/73560.73564.

P. Melliès. Segal condition meets computational effects. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 150–159, 2010. doi: 10.1109/LICS.2010.46. URL http://dx.doi.org/10.1109/LICS.2010.46.

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4. URL http://dx.doi.org/10.1016/0890-5401(91)90052-4.

D. Orchard. Should i use a monad or a comonad? Technical report, 2012.

D. A. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: Declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 15–24, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-859-9. doi: 10.1145/1708046.1708053. URL http://doi.acm.org/10.1145/1708046.1708053.

T. Petricek, D. Orchard, and A. Mycroft. Coeffects: unified static analysis of context-dependence. In *Proceedings of International Conference on Automata, Languages, and Programming - Volume Part II*, ICALP 2013.

G. D. Plotkin and J. Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, pages 342–356, 2002. doi: 10.1007/3-540-45931-6_24. URL http://dx.doi.org/10.1007/3-540-45931-6_24.

G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 80–94, 2009. doi: 10.1007/978-3-642-00590-9_7. URL http://dx.doi.org/10.1007/978-3-642-00590-9_7.

G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi: 10.2168/LMCS-9(4:23)2013. URL http://dx.doi.org/10.2168/LMCS-9(4:23)2013.

M. Pretnar. Inferring algebraic effects. *Logical Methods in Computer Science*, 10(3), 2014. doi: 10.2168/LMCS-10(3:21)2014. URL http://dx.doi.org/10.2168/LMCS-10(3:21)2014.

L. Schröder and T. Mossakowski. Generic exception handling and the Java monad. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, pages 443–459. Springer, 2004. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3116&spage=443.

N. Tzevelekos. *Nominal game semantics*. PhD thesis, 2008. CS-RR-09-18.

T. Uustalu and V. Vene. The essence of dataflow programming. In *Central European Functional Programming School, First Summer School, CEFP 2005, Budapest, Hungary, July 4-15, 2005, Revised Selected Lectures*, pages 135–167, 2005. doi: 10.1007/11894100_5. URL http://dx.doi.org/10.1007/11894100_5.

T. Uustalu and V. Vene. Comonadic notions of computation. *Electron. Notes Theor. Comput. Sci.*, 203(5): 263–284, June 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.05.029. URL http://dx.doi.org/10.1016/j.entcs.2008.05.029.

P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143169. URL http://doi.acm.org/10.1145/143165.143169.