Contract Monitoring and Call-by-name Evaluation

— Extended Version, TR 243, Oct. 2008 —

Markus Degen Peter Thiemann Stefan Wehr

Institut für Informatik, Universität Freiburg, Germany

{degen,thiemann,wehr}@informatik.uni-freiburg.de

Abstract

Contracts are a proven tool in software development. They provide specifications for operations that may be statically verified or dynamically validated by contract monitoring.

Contract monitoring for strict languages has a by now established theoretical basis. This article investigates the properties of contract monitoring for languages with call-by-name evaluation. We study two different styles of contract monitoring: eager monitoring enforces a contract when it is demanded, possibly evaluating expressions not touched by the user code, whereas delayed monitoring only proceeds as far as the user code itself can observe.

In each case, an effect system ensures that contract monitoring does not change the meaning of a program and guarantees that contract monitoring is idempotent. Our formalization brings forward semantic reasons that favor delayed monitoring for a call-by-name language and comes with a Haskell implementation.

1. Introduction

Design by contract [16] is a methodology for constructing correct software. Each operation is associated with a contract that defines two assertions, a *precondition* and a *postcondition*, for the operation. The contract is fulfilled if the usual partial correctness condition is true: If the input meets the precondition and the operation produces output, then the output is obliged to meet the postcondition. A program run *violates* a contract if any of the pre- and postconditions of the operations involved is false. Thus, a contract provides a (partial) specification of an operation that every implementation of the operation must fulfill.

While contracts can be verified statically, in practice they are often enforced dynamically using *contract monitoring* (cf. Eiffel [15, 14], Java [1, 13], Scheme [20], or Haskell [12]): The implementation of an operation with monitoring checks the precondition before performing its computation and checks the postcondition before returning to its caller. If the precondition of the operation is false, then it raises an exception blaming its caller. Conversely, if the postcondition does not hold, then the operation blames itself.

The semantics of contract monitoring is intricate, and its correct and complete implementation is non-trivial [7] (see Section 5 for further discussion on related work). From a practical point of view, contract monitoring should guarantee meaning preservation (MP) and behave idempotently (IP):

MP: If a program run with contract monitoring enabled has no contract violations, then disabling contract monitoring should not change its meaning.

IP: Applying a contract multiple times is equivalent to applying it once.

The MP property ensures that developers may enable contract monitoring for a test version of their software and safely disable contract monitoring for the release version, without running the risk that the test and the release version behave differently. The IP property ensures a meaningful notion of contract composition.

In the context of call-by-value evaluation, there is only one useful and sensible mode of contract monitoring. This mode corresponds to its implementation in Eiffel, Java, and Scheme and is the one we just described.

In the context of call-by-name evaluation, there are at least two options, *eager monitoring* and *delayed monitoring*. Eager monitoring enforces an assertion when it is demanded. That is, it checks the precondition when the function demands its argument and it checks the postcondition when the caller demands the function's result. This eager strategy sometimes leads to undesirable behavior which violates the IP property as pointed out by Hinze et al. [12].

We have developed a formalization which precisely pinpoints where eager monitoring imposes too many restrictions on expressions subject to a contract. Hence, we propose delayed monitoring as an alternative form of contract monitoring for languages with call-by-name evaluation. Delayed monitoring places no restrictions on expressions subject to a contract and enjoys the MP and IP properties. It defers enforcement of an assertion until all values that it depends on are evaluated by user code. Thus, monitoring only proceeds as far as the user code itself can observe. Violations that the user code cannot observe, yet, are considered to be invisible. Perhaps surprisingly, this delayed interpretation has a logical foundation: While call-by-value monitoring checks properties according to classical logic, lazy monitoring relies on a three-valued logic.

Contributions

We have developed a semantic framework for specifying and comparing contract monitoring in functional languages. The basis of the framework is an extended version of Moggi's monadic metalanguage [17] with a fixed monad providing for nontermination, mutable state, and exceptions[1] and an effect system for keeping track of the effects.

We have developed the semantics of two styles of contract monitoring for impure functional languages with call-by-name evaluation, eager and delayed monitoring. Both are defined by translation into the metalanguage.

The semantics enables us to formally prove (or disprove) the MP and IP properties. The semantics also explains and helps fixing a problem with eager monitoring observed by Hinze et al. [12]. We define and prove correct a criterion that fixes the problem by imposing a suitable typing discipline.

[1] Usually, call-by-name languages provide non-termination as the only effect. However, there is often a back door that allows other effects to creep in. In Haskell, this back door is called `unsafePerformIO`.

$$
\begin{aligned}
S ::= {} & x \mid () \mid \texttt{rec}\, f(x) = S \mid S\, S \mid (S, S) \mid \texttt{in}_i\, S \\
& \mid \texttt{case}\, S\, \texttt{of}\, (x_1, x_2) : S \mid \texttt{case}\, S\, \texttt{of}\, \texttt{in}_1\, x : S; \texttt{in}_2\, x : S \\
& \mid \texttt{raise} \mid \texttt{try}\, S\, \texttt{catch}\, S \\
& \mid \texttt{ref}_\rho\, S \mid \texttt{get}\, S \mid \texttt{set}\, S\, S \mid \texttt{seq}\, x \Leftarrow S; S \mid !C\, S \\
C ::= {} & ?x.S \mid H^x_\rightarrow\, C\, C \mid H_\times\, C\, C \mid H_+\, C\, C \mid \&\, C\, C
\end{aligned}
$$

Figure 1: Syntax of the source language. $x, f \in \mathit{Var}$ and $\rho \in \mathit{Reg}$ drawn from countably infinite sets of variables and region constants.

Finally, we present a prototype implementation of delayed monitoring in Haskell.[2]

Overview

The basis of our investigation is an impure language with contracts introduced in Section 2. For this language, we provide two translations into a version of Moggi's metalanguage. The translation to the metalanguage makes all effects explicit, thus creating a framework for formal reasoning about contract monitoring. Section 3 covers the variant with eager monitoring whereas Section 4 deals with delayed monitoring. We prove that all variants fulfill MP and IP. Section 4 also presents a novel Haskell implementation of delayed monitoring. We discuss related work in Section 5. Section 6 concludes and gives pointers to future work.

2. A Language with Contracts

We study $\lambda^{?!}$, a simply-typed lambda calculus extended with contracts. After defining its syntax, we present some example contracts and discuss their interpretation. Meaning preservation and idempotency fail to hold for some of the example terms because the syntax of $\lambda^{?!}$ admits arbitrary side-effects in certain contracts. Hence, we propose a type and effect system [11, 19] that enables us to restrict the effects of a predicate. Section 3 formally specifies the system for eager contract monitoring, Section 4 provides the system for delayed contract monitoring. Both systems derive from the transformation to a common typed metalanguage.

We also mention the call-by-value interpretation of contracts as an intuitive reference but have omitted the formalization of its semantics and its type system because its construction is tedious but standard and the formalization does not yield any unexpected insights.

2.1 Syntax

Figure 1 defines the syntax of $\lambda^{?!}$. Expressions S comprise variables, unit values, introduction and elimination forms for recursive functions, pairs, and sums, as well as exceptions and references. Reference initialization, written $\texttt{ref}_\rho\, S$, specifies the region ρ in which the reference is to be allocated. The expression $\texttt{seq}\, x \Leftarrow S_1; S_2$ evaluates S_1, binds the result to x, and then evaluates S_2. Contract enforcement, $!C\, S$, asserts that S fulfills contract C. We call S the *subject* of C. Contract enforcement either throws a special *blame exception* if the contract fails or it returns the expression's value if the contract holds.

There are five forms of contracts that serve as some sort of filter on their subject. A *predicate contract* $?x.S$ evaluates S with the binding of x to the subject. It returns the subject if S is true. If S is false, it raises a contract exception. Otherwise, the contract behaves just like S: it may raise an exception or it may not terminate.

A *function contract* $H^x_\rightarrow\, C_1\, C_2$ confines v to be a function with precondition C_1 and postcondition C_2. It binds the variable x to the function's argument so that the postcondition C_2 can refer to it. A *pair contract* $H_\times\, C_1\, C_2$ (*sum contract* $H_+\, C_1\, C_2$) restricts the components of a pair (sum) according to C_1 and C_2. We use the term *structural contract* to refer to either a function, a pair, or a sum contract.

Finally, $\&\, C_1\, C_2$ denotes the *conjunction* of C_1 and C_2, which is realized as a composition that first applies C_2 to the subject and then C_1 to the result.

Notation We write $\lambda x.S$ instead of $\texttt{rec}\, f(x) = S$ if f is not free in S and let $\equiv$ indicate syntactic equality. We set $\mathsf{true} \equiv \texttt{in}_1\, ()$ and $\mathsf{false} \equiv \texttt{in}_2\, ()$ to define the conditional: $\mathsf{if}\, S\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \equiv \texttt{case}\, S\, \texttt{of}\, \texttt{in}_1\, x_1 : e_1; \texttt{in}_2\, x_2 : e_2$ where x_1, x_2 are fresh.

2.2 Examples

Here are some contract terms written in $\lambda^{?!}$ extended with integers and some primitives. They serve as running examples which are discussed throughout the paper. Let $\mathscr{C}_{id} \equiv ?x.\mathsf{true}$ be the identity contract.

1. The predicate contract $\mathscr{C}_1 \equiv ?x.\texttt{case}\, x\, \texttt{of}\, (x_1, x_2) : x_1 < x_2$ asserts that the first component of a pair is smaller than the second.
2. The function contract $\mathscr{C}_2 \equiv H^x_\rightarrow\, (?x.x \geq 0)\, (?y.y * y = x)$ guarantees that a function computes the square root unless its argument is negative.
3. Suppose that π is an expression the evaluation of which may not terminate. Enforcing the pair contract
$$\mathscr{C}_3 \equiv H_\times\, (?x.\mathsf{if}\, x = 42\, \mathsf{then}\, \texttt{raise}\, \mathsf{else}\, \mathsf{true})\, (?x.\pi)$$
raises an exception on pairs with first component 42 (the exception is neither caught nor converted to a contract failure) and may not terminate unless the predicate π terminates.
4. The sum contract $\mathscr{C}_4 \equiv H_+\, (?x.\texttt{set}\, y\, (\texttt{get}\, y + 1))\, \mathscr{C}_{id}$ uses a global reference cell y for counting how often the contract is enforced on an expression of the form $\texttt{in}_1\, S$.
5. The contract $\mathscr{C}_5 \equiv \&\, \mathscr{C}_3'\, \mathscr{C}_1$ where $\mathscr{C}_3' \equiv H_\times\, (?x.x < 0)\, \mathscr{C}_{id}$ is a conjunction which asserts that the first component of a pair is negative and smaller than the second.

The rest of this section uses the examples to compare contract monitoring for call-by-value with eager contract monitoring for call-by-name on an intuitive level. The discussion focuses on the fulfillment of the MP and IP properties. The observations for the call-by-value setting remain on an intuitive basis[3] whereas Section 3 provides the formal underpinnings for the call-by-name setting with eager monitoring.

1. Under call-by-value evaluation, $\mathscr{C}_1$ enjoys MP and IP. Under call-by-name evaluation with eager monitoring, neither MP nor IP hold unless the effect of the pair's components is restricted to reading reference cells. (Nontermination is also ruled out by this effect restriction.) Consider applying the contract to the pair $(\texttt{set}\, y\, (\texttt{get}\, y + 1), 0)$. In a context that only extracts the second component, the write operation takes place if monitoring is enabled, but is ignored without it. Moreover, enforcing the contract once leaves a different value in y than enforcing it twice. (Under call-by-need evaluation, which we do not consider in this article, the contract would fulfill IP.)
2. Under call-by-value evaluation, $\mathscr{C}_2$ enjoys MP and IP because its pre- and postconditions terminate and do not have side-effects. Under call-by-name evaluation, the situation is very similar to example 1: If f is the function resulting from attaching the contract to some function g, then MP and IP only holds if the effect of f's argument is restricted to reading reference cells (because the contract's postcondition evaluates the argument even

[2] The implementation is available from `http://proglang.informatik.uni-freiburg.de/projects/contracts/`.

[3] as this paper omits the call-by-value formalization.

if g does not). Section 4 investigates delayed contract monitoring for call-by-name evaluation as a remedy for the situations discussed in examples 1 and 2.

3. Idempotency of the pair contract $\mathscr{C}_3$ should be obvious. However, it does not always preserve the meaning of programs. With contract monitoring enabled, accessing the first component may trigger an exception and accessing the second component may result in a non-terminating computation. Neither outcome is possible if contract monitoring is disabled.

   An alternative semantics of contracts might interpret raising of an exception or nontermination of a contract as a contract failure, in which case MP would be trivially true.

4. The sum contract $\mathscr{C}_4$ neither preserves the meaning nor is it idempotent because it writes to a global reference cell.

5. As pointed out by Hinze et al. [12], contract conjunction is, in general, not idempotent under a call-by-name evaluation strategy with eager monitoring.

   Consider the expression

   $$\mathtt{case}\ !\mathscr{C}_5\ (1,2)\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

   It is equivalent to

   $$\mathtt{case}\ !\mathscr{C}_3'\ (!\mathscr{C}_1\ (1,2))\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

   which evaluates to false because $\mathscr{C}_1$ is imposed first and succeeds. Then the pair contract $\mathscr{C}_3'$ is imposed but never checked because the pair's components are not evaluated.

   However, evaluating the expression

   $$\mathtt{case}\ !\mathscr{C}_5\ (!\mathscr{C}_5\ (1,2))\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

   which is equivalent to

   $$\mathtt{case}\ !\mathscr{C}_3'\ (!\mathscr{C}_1\ (!\mathscr{C}_3'\ (!\mathscr{C}_1\ (1,2))))\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

   signals a contract violation because enforcing $\mathscr{C}_1$ a second time triggers the evaluation of the formerly unchecked $\mathscr{C}_3'$, which in turn raises the blame exception. See Section 3.4 and Section 3.5 for further discussion.

   For call-by-value, we can construct an analogous example to show that composition is not idempotent, either. Let

   $$\begin{array}{lcl} V_1 & = & ?x.\mathtt{case}\ x\ \mathtt{of}\ (x_1,x_2) : x_1(0) < x_2(0) \\ V_2 & = & H_\times\ (H^y_\to\ \mathscr{C}_{id}\ (?x.x<0))\ \mathscr{C}_{id} \end{array}$$

   and observe that $\mathtt{case}\ !(\&\ V_2\ V_1)\ ((\lambda z.1),(\lambda z.2))\ \mathtt{of}\ (x_1,x_2) :$ false evaluates to false because enforcing V_1 leaves the subject unchanged and then V_2 imposes a function contract on its first component, which is never exercised. On the other hand, $\mathtt{case}\ !(\&\ V_2\ V_1)\ (!(\&\ V_2\ V_1)\ ((\lambda z.1),(\lambda z.2)))\ \mathtt{of}\ (x_1,x_2)\ :$ false raises a blame exception because the second application of V_1 forces exactly the function contract imposed by V_2, which $\lambda z.1$ and $\lambda z.2$ do not fulfill.

3. Eager Contract Monitoring

This section defines a type and effect system for eager contract monitoring. Next, it defines the the semantics of eager contract monitoring by translation to an extended version of Moggi's monadic metalanguage, $\lambda^{ML+\times}$. Definition and typing of this metalanguage are standard and its extension is fairly straightforward. For completeness Appendix A contains its formal definition and the properties needed to prove the results of this section. After presenting the translation and establishing its basic properties, the section goes on to discuss meaning preservation, contract conjunction and idempotency of contracts.

3.1 Static Semantics

Figure 2 contains a type and effect system for $\lambda^{?!}$ with call-by-name evaluation and eager contract monitoring. Effects ϵ include nontermination ($\perp$), exceptions (e), reading ($\mathsf{r}(\rho)$) and writing ($\mathsf{w}(\rho)$) of references in region ρ, and the union of two effects ($\epsilon \cup \epsilon$). Regions are only used in Section 4 for delayed monitoring, they are of no consequence for eager monitoring. An exception may be a user exception (u) or blame for a contract violation (b). The union operator $\cup$ is associative, commutative, idempotent, and has $\emptyset$ as a unit. We write $\epsilon' \subseteq \epsilon$ when $\epsilon \cup \epsilon' = \epsilon$ and $\cup_\rho \mathsf{r}(\rho)$ for $\{\mathsf{r}(\rho) \mid \rho \in \mathit{Reg}\}$.

Every type is paired with an effect. Thus, there are two categories of types: the category U contains the usual type constructors applied to T-types, whereas T is a U-type annotated with effect ϵ. The U-type bool is then defined as $*^\emptyset + *^\emptyset$. A typing environment A maps variables to T-types.

The subtyping relation $T \leq T'$ is standard. The judgment $A \vdash_n S : T$ states that expression S has type T. Most of the typing rules for expressions are adaptations of the rules for the simply-typed lambda calculus. The rule for recursive functions implements a simple termination detection by using the nontermination effect $\perp$ in the type assumption [2]. Function application collects the effects for evaluating the function and for running it, the argument's effect is not (directly) considered. The rule for catching exceptions only allows for catching of user exceptions, which are masked out of the effect of the try-expression. Reading a reference in region ρ has the effect $\mathsf{r}(\rho)$ combined with the effect for evaluating the reference and its contents, whereas writing a reference does not evaluate the expression written. The rule for sequential composition indicates that $\mathtt{seq}\ x \Leftarrow S_1; S_2$ evaluates S_1 before evaluating S_2: The effect of S_1 is part of the expression's effect and the binding of x does not carry an effect anymore. Applying a contract is similar to a function application, except that there is no evaluation of the contract so that only the effect of running the contract remains.

The judgment $A \vdash_n C \rhd T \rightsquigarrow T'$ asserts that C is a contract for expressions of type T which results in an expression of type T' once the contract is enforced. The rule for a predicate contract $?x.S$, types S with the assumption that evaluating x is effect free. The dynamic semantics (see Section 3) forces evaluation of x before checking the predicate S. The rule for typing a conjunction $\&\ C_1\ C_2$ ensures that the order in which C_1 and C_2 are enforced does not influence typability. In Section 3.5 we show that the order independence of the typing implies that C_1 and C_2 commute, which in turn implies idempotency of $\&\ C_1\ C_2$.

Examples

A review of the example contracts from Section 2.2 yields that contracts $\mathscr{C}_3, \mathscr{C}_4$, and $\mathscr{C}_5$ are rejected (they would violate MP or IP) and that $\mathscr{C}_1$ ($\mathscr{C}_2$) can be enforced only on pairs (functions) whose components (arguments) have no effect other than $\cup_\rho \mathsf{r}(\rho)$.

1. $A \vdash_n \mathscr{C}_1 \rhd (\mathtt{int}^{\epsilon_1} \times \mathtt{int}^{\epsilon_2})^{\epsilon'} \rightsquigarrow (\mathtt{int}^{\epsilon_1} \times \mathtt{int}^{\epsilon_2})^{\epsilon_1 \cup \epsilon_2 \cup \epsilon' \cup \mathsf{b}}$ with $\epsilon_1 \cup \epsilon_2 \subseteq \cup_\rho \mathsf{r}(\rho)$. The typing rule for a predicate contract first strips the top-level effect of the incoming pair (as this effect happens anyway when the pair is evaluated). Thus eliminating the pair has no effect, but evaluating $x_1 < x_2$ has effect $\epsilon_1 \cup \epsilon_2$, which is thus the effect of the entire boolean expression. Finally, the predicate rule imposes the read restriction on exactly this effect.

   The typing reveals that this contract cannot be applied to a pair $(S,0)$ for a possibly diverging S because $(S,0)$ would have type $(\mathtt{int}^\perp \times \mathtt{int}^\emptyset)^\emptyset$ and $\perp \not\subseteq \cup_\rho \mathsf{r}(\rho)$. Section 4 addresses this shortcoming.

Effects

$$\epsilon ::= \emptyset \mid \bot \mid \mathsf{e} \mid \mathsf{r}(\rho) \mid \mathsf{w}(\rho) \mid \epsilon \cup \epsilon \qquad \mathsf{e} ::= \mathsf{u} \mid \mathsf{b}$$

Types and environments

$$U ::= * \mid \mathtt{ref}_\rho\, T \mid T \times T \mid T + T \mid T \to T \qquad T ::= U^\epsilon \qquad A ::= \cdot \mid A, x : T$$

Subtyping

$$* \le * \quad \mathtt{ref}_\rho\, T \le \mathtt{ref}_\rho\, T \quad \frac{U_1 \le U_1' \quad U_2 \le U_2'}{U_1 \times\!\!\!+ U_2 \le U_1' \times\!\!\!+ U_2'} \quad \frac{U_1 \le U_1' \quad U_2 \le U_2'}{U_1' \to U_2 \le U_1 \to U_2'} \quad \frac{U \le U' \quad \epsilon \subseteq \epsilon'}{U^\epsilon \le U'^{\epsilon'}}$$

Typing rules for expressions

$$A \vdash_n () : *^\emptyset \qquad \frac{A(x) = T}{A \vdash_n x : T} \qquad \frac{A, f : (T \to U^{\epsilon \cup \bot})^\emptyset, x : T \vdash_n S : U^\epsilon}{A \vdash_n \mathtt{rec}\, f(x) = S : (T \to U^\epsilon)^\emptyset} \qquad \frac{A \vdash_n S_1 : (T \to U^\epsilon)^{\epsilon'} \quad A \vdash_n S_2 : T}{A \vdash_n S_1\, S_2 : U^{\epsilon \cup \epsilon'}}$$

$$\frac{A \vdash_n S_1 : T_1 \quad A \vdash_n S_2 : T_2}{A \vdash_n (S_1, S_2) : (T_1 \times T_2)^\emptyset} \qquad \frac{A \vdash_n S : (T_1 \times T_2)^{\epsilon'} \quad A, x_1 : T_1, x_2 : T_2 \vdash_n S' : U^\epsilon}{A \vdash_n \mathtt{case}\, S\, \mathtt{of}\, (x_1, x_2) : S' : U^{\epsilon \cup \epsilon'}} \qquad \frac{A \vdash_n S : T_i}{A \vdash_n \mathtt{in}_i\, S : (T_1 + T_2)^\emptyset}$$

$$\frac{A \vdash_n S : (T_1 + T_2)^\epsilon \; A, x_1 : T_1 \vdash_n S_1 : U^{\epsilon_1} \; A, x_2 : T_2 \vdash_n S_2 : U^{\epsilon_2}}{A \vdash_n \mathtt{case}\, S\, \mathtt{of}\, \mathtt{in}_1\, x_1 : S_1; \mathtt{in}_2\, x_2 : S_2 : U^{\epsilon \cup \epsilon_1 \cup \epsilon_2}} \qquad A \vdash_n \mathtt{raise} : U^{\mathsf{u}} \qquad \frac{A \vdash_n S : U^\epsilon \quad A \vdash_n S' : U^{\epsilon'}}{A \vdash_n \mathtt{try}\, S\, \mathtt{catch}\, S' : U^{(\epsilon \setminus \mathsf{u}) \cup \epsilon'}}$$

$$\frac{A \vdash_n S_1 : (\mathtt{ref}_\rho\, U^\epsilon)^{\epsilon'} \quad A \vdash_n S_2 : U^\epsilon}{A \vdash_n \mathtt{set}\, S_1\, S_2 : *^{\mathsf{w}(\rho) \cup \epsilon'}} \qquad \frac{A \vdash_n S : T}{A \vdash_n \mathtt{ref}_\rho\, S : (\mathtt{ref}_\rho\, T)^\emptyset} \qquad \frac{A \vdash_n S : (\mathtt{ref}_\rho\, U^\epsilon)^{\epsilon'}}{A \vdash_n \mathtt{get}\, S : U^{(\mathsf{r}(\rho) \cup \epsilon \cup \epsilon')}}$$

$$\frac{A \vdash_n S_1 : U_1^{\epsilon_1} \;\; A, x : U_1^\emptyset \vdash_n S_2 : U_2^{\epsilon_2}}{A \vdash_n \mathtt{seq}\, x \Leftarrow S_1; S_2 : U_2^{\epsilon_1 \cup \epsilon_2}} \qquad \frac{A \vdash_n S : T' \quad T' \le T}{A \vdash_n S : T} \qquad \frac{A \vdash_n C \triangleright T \leadsto T' \quad A \vdash_n S : T}{A \vdash_n\, !C\, S : T'}$$

Typing rules for contracts

$$\frac{A, x : U^\emptyset \vdash_n S : \mathsf{bool}^\epsilon \quad \epsilon \subseteq \cup_\rho \mathsf{r}(\rho)}{A \vdash_n ?x.S \triangleright U^{\epsilon'} \leadsto U^{\epsilon \cup \epsilon' \cup \mathsf{b}}} \qquad \frac{A \vdash_n C_1 \triangleright T_1 \leadsto T_1' \quad A, x : T_1 \vdash_n C_2 \triangleright T_2 \leadsto T_2'}{A \vdash_n H^x_\to\, C_1\, C_2 \triangleright (T_1' \to T_2)^{\epsilon'} \leadsto (T_1 \to T_2')^{\epsilon'}}$$

$$\frac{A \vdash_n C_1 \triangleright T_1 \leadsto T_1' \quad A \vdash_n C_2 \triangleright T_2 \leadsto T_2'}{A \vdash_n H_{\times\!\!\!+}\, C_1\, C_2 \triangleright (T_1 \times\!\!\!+ T_2)^\epsilon \leadsto (T_1' \times\!\!\!+ T_2')^\epsilon} \qquad \frac{A, x : T \vdash_n\, !C_1\, (!C_2\, x) : T' \quad A, x : T \vdash_n\, !C_2\, (!C_1\, x) : T'}{A \vdash_n \&\, C_1\, C_2 \triangleright T \leadsto T'}$$

Figure 2: Type and effect system for call-by-name with eager contract monitoring.

2. $A \vdash_n \mathscr{C}_2 \triangleright (\mathtt{int}^{\epsilon \cup \mathsf{b}} \to \mathtt{int}^{\epsilon'})^{\epsilon''} \leadsto (\mathtt{int}^\epsilon \to \mathtt{int}^{\epsilon \cup \epsilon' \cup \mathsf{b}})^{\epsilon''}$ with $\epsilon \subseteq \cup_\rho \mathsf{r}(\rho)$.

   This typing points to the same shortcoming of eager contract monitoring. Again, because $\bot \not\subseteq \cup_\rho \mathsf{r}(\rho)$ a function $!\mathscr{C}_2\, f$ cannot be applied to a potentially nonterminating argument of type $\mathtt{int}^\bot$.
3. No type is derivable for $\mathscr{C}_3$ because the contract for the first component raises an exception.
4. No type is derivable for $\mathscr{C}_4$ because the contract for the first summand writes to memory.
5. No type is derivable for $\mathscr{C}_5$: we have $A \vdash_n \mathscr{C}_3' \triangleright (\mathtt{int}^{\epsilon_1} \times \mathtt{int}^{\epsilon_2})^\epsilon \leadsto (\mathtt{int}^{\epsilon_1 \cup \mathsf{b}} \times \mathtt{int}^{\epsilon_2 \cup \mathsf{b}})^\epsilon$ but no type is derivable for $!\mathscr{C}_1\, (!\mathscr{C}_3'\, x)$ because $\mathscr{C}_1$ does not accept pairs of type $(\mathtt{int}^{\epsilon_1 \cup \mathsf{b}} \times \mathtt{int}^{\epsilon_2 \cup \mathsf{b}})^\epsilon$.

3.2 Dynamic Semantics

Figure 3 defines the dynamic semantics of $\lambda^{?!}$ under call-by-name evaluation and eager contract monitoring by translation to an extended version of Moggi's monadic metalanguage, $\lambda^{ML+\times}$. The bind operator of $\lambda^{ML+\times}$ is written $\mathtt{let}\, x \Leftarrow e\, \mathtt{in}\, e'$ and $[e]$ denotes the return operator. Moreover, $\lambda^{ML+\times}$ provides constants for constructing recursive functions (`rec`), for raising and handling exceptions ($\mathtt{raise}_{\mathsf{e}}$, $\mathtt{try\_catch}_{\mathsf{e}}$), as well as for allocating, reading, and writing reference cells ($\mathtt{ref}_\rho$, `get`, `set`).

The translation from $\lambda^{?!}$ to $\lambda^{ML+\times}$ consists of two mutually recursive parts, $\mathcal{S}_n[\![]\!]$, for translating expressions S, their types T, U, and type assumptions, and $\mathcal{C}_n[\![]\!]$ for translating contracts C. The translation of expressions, types, and type assumptions is a standard call-by-name monadic translation. The translation of contracts requires some explanation.

As each translated expression has a computation type $\mathbf{T}_\epsilon(t)$, the translation of a contract is a function that maps a subject computation to another computation. Contract enforcement just applies this function to the subject. The contract function is *not* wrapped in a computation type.

The translation of a predicate contract first evaluates the subject, thus performing the effect of its evaluation once. This construction is required for two reasons. First, performing this evaluation is correct and does not affect the semantics as the evaluation of the contract is caused by a demand for the subject. Second, it avoids potential repetition of the evaluation of the subject (and its effect) during evaluation of the predicate itself.

Next, the resulting value y is wrapped in a trivial computation and passed to the translation of the predicate. If the result of the predicate is `true`, then the result of the contract is the wrapped value, otherwise it raises the blame exception.

The translations of the structural contracts all follow the same pattern. Each evaluates its subject, then instruments the resulting value to enforce the subcontracts, and finally wraps the result in a trivial computation. The function contract $H^y_\to\, C_1\, C_2$ returns

$$
\begin{array}{rcl}
\mathcal{S}_n[\![x]\!] & \equiv & x \\
\mathcal{S}_n[\![()]\!] & \equiv & [()] \\
\mathcal{S}_n[\![\mathtt{rec}\, f(x) = S]\!] & \equiv & [\mathtt{rec}\,\lambda f.\lambda x.(\lambda f.\mathcal{S}_n[\![S]\!])\,[f]] \\
\mathcal{S}_n[\![S_1\, S_2]\!] & \equiv & \mathtt{let}\, y \Leftarrow \mathcal{S}_n[\![S_1]\!]\,\mathtt{in}\, y\,(\mathcal{S}_n[\![S_2]\!]) \\
\mathcal{S}_n[\![\mathtt{seq}\, x \Leftarrow S_1; S_2]\!] & \equiv & \mathtt{let}\, y \Leftarrow \mathcal{S}_n[\![S_1]\!]\,\mathtt{in}\,(\lambda x.\mathcal{S}_n[\![S_2]\!])\,[y] \\
\mathcal{S}_n[\![(S_1, S_2)]\!] & \equiv & [(\mathcal{S}_n[\![S_1]\!], \mathcal{S}_n[\![S_2]\!])] \\
\mathcal{S}_n[\![\mathtt{case}\, S\,\mathtt{of}\,(x_1, x_2) : S_0]\!] & \equiv & \mathtt{let}\, y \Leftarrow \mathcal{S}_n[\![S]\!]\,\mathtt{in}\,\mathtt{case}\, y\,\mathtt{of}\,(x_1, x_2) : \mathcal{S}_n[\![S_0]\!] \\
\mathcal{S}_n[\![\mathtt{in}_i\, S]\!] & \equiv & [\mathtt{in}_i\,(\mathcal{S}_n[\![S]\!])] \\
\mathcal{S}_n[\![\mathtt{case}\, S\,\mathtt{of}\,\mathtt{in}_1\, x_1 : S_1; \mathtt{in}_2\, x_2 : S_2\ ]\!] & \equiv & \mathtt{let}\, y \Leftarrow \mathcal{S}_n[\![S]\!]\,\mathtt{in}\,\mathtt{case}\, y\,\mathtt{of}\,\mathtt{in}_1\, x_1 : \mathcal{S}_n[\![S_1]\!]; \mathtt{in}_2\, x_2 : \mathcal{S}_n[\![S_2]\!] \\
\\
\mathcal{S}_n[\![\mathtt{raise}\,]\!] & \equiv & \mathtt{raise}_{\mathsf{u}} \\
\mathcal{S}_n[\![\mathtt{try}\, S_1\,\mathtt{catch}\, S_2]\!] & \equiv & \mathtt{try}\,\mathcal{S}_n[\![S_1]\!]\,\mathtt{catch}_{\mathsf{u}}.\mathcal{S}_n[\![S_2]\!] \\
\mathcal{S}_n[\![\mathtt{ref}_\rho\, S]\!] & \equiv & \mathtt{ref}_\rho\,(\mathcal{S}_n[\![S]\!]) \\
\mathcal{S}_n[\![\mathtt{get}\, S]\!] & \equiv & \mathtt{let}\, y \Leftarrow \mathcal{S}_n[\![S]\!]\,\mathtt{in}\,\mathtt{let}\, x \Leftarrow \mathtt{get}\, y\,\mathtt{in}\, x \\
\mathcal{S}_n[\![\mathtt{set}\, S_1\, S_2]\!] & \equiv & \mathtt{let}\, y \Leftarrow \mathcal{S}_n[\![S_1]\!]\,\mathtt{in}\,\mathtt{set}\, y\,(\mathcal{S}_n[\![S_2]\!]) \\
\\
\mathcal{S}_n[\![!C\, S]\!] & \equiv & \mathcal{C}_n[\![C]\!]\,(\mathcal{S}_n[\![S]\!]) \\
\mathcal{C}_n[\![?x.S]\!] & \equiv & \lambda z.\mathtt{let}\, y \Leftarrow z\,\mathtt{in}\,(\lambda x.\mathtt{let}\, x_S \Leftarrow \mathcal{S}_n[\![S]\!]\,\mathtt{in}\,\mathsf{if}\, x_S\,\mathsf{then}\, x\,\mathsf{else}\,\mathtt{raise}_{\mathsf{b}}\,)\,[y] \\
\mathcal{C}_n[\![H^x_\to\, C_1\, C_2]\!] & \equiv & \lambda z.\mathtt{let}\, y \Leftarrow z\,\mathtt{in}\,[\lambda x.\mathcal{C}_n[\![C_2]\!]\,(y\,(\mathcal{C}_n[\![C_1]\!]\, x))] \\
\mathcal{C}_n[\![H_\times\, C_1\, C_2]\!] & \equiv & \lambda z.\mathtt{let}\, y \Leftarrow z\,\mathtt{in}\,\mathtt{case}\, y\,\mathtt{of}\,(x_1, x_2) : [(\mathcal{C}_n[\![C_1]\!]\, x_1, \mathcal{C}_n[\![C_2]\!]\, x_2)] \\
\mathcal{C}_n[\![H_+\, C_1\, C_2]\!] & \equiv & \lambda z.\mathtt{let}\, y \Leftarrow z\,\mathtt{in}\,\mathtt{case}\, y\,\mathtt{of}\,\mathtt{in}_1\, x_1 : [\mathtt{in}_1\,(\mathcal{C}_n[\![C_1]\!]\, x_1)]; \mathtt{in}_2\, x_2 : [\mathtt{in}_2\,(\mathcal{C}_n[\![C_2]\!]\, x_2)] \\
\mathcal{C}_n[\![\&\, C_1\, C_2]\!] & \equiv & \lambda z.\mathcal{C}_n[\![C_1]\!]\,(\mathcal{C}_n[\![C_2]\!]\, z) \\
\\
\mathcal{S}_n[\![*]\!] & \equiv & * \\
\mathcal{S}_n[\![\mathtt{ref}_\rho\, T]\!] & \equiv & \mathtt{ref}_\rho\,\mathcal{S}_n[\![T]\!] \\
\mathcal{S}_n[\![T_1 \times T_2]\!] & \equiv & \mathcal{S}_n[\![T_1]\!] \times \mathcal{S}_n[\![T_2]\!] \\
\mathcal{S}_n[\![T_1 + T_2]\!] & \equiv & \mathcal{S}_n[\![T_1]\!] + \mathcal{S}_n[\![T_2]\!] \\
\mathcal{S}_n[\![T_1 \to T_2]\!] & \equiv & \mathcal{S}_n[\![T_1]\!] \to \mathcal{S}_n[\![T_2]\!] \\
\\
\mathcal{S}_n[\![U^\epsilon]\!] & \equiv & \mathbf{T}_\epsilon(\mathcal{S}_n[\![U]\!]) \\
\\
\mathcal{S}_n[\![\cdot]\!] & \equiv & \cdot \\
\mathcal{S}_n[\![A, x : T]\!] & \equiv & \mathcal{S}_n[\![A]\!], x : \mathcal{S}_n[\![T]\!]
\end{array}
$$

Figure 3: Call-by-name translation of $\lambda^{?!}$ to $\lambda^{ML+\times}$ with eager contract monitoring.

the composition of C_2, the original function, and C_1. The pair contract $H_\times\, C_1\, C_2$ returns the pair constructed of the components of the original pair after application of C_1 and C_2 to them. The sum contract $H_+\, C_1\, C_2$ returns a tagged value after untagging the original value and applying C_1 or C_2 to it, depending on the original tag.

As advertised before, the translation of the conjunction contract $\&\, C_1\, C_2$ is the composition of the translations of C_1 and C_2 so that C_2 is applied first to the subject. The choice of enforcing C_2 first is somewhat arbitrary. However, it does not influence the dynamic semantics because our type system rules out the conjunction of non-commuting contracts (see Section 3.4).

The translation passes the basic metatheoretical sanity check in that it preserves typing.

Theorem 1 (Translation from $\lambda^{?!}$ to $\lambda^{ML+\times}$ Preserves Typing)**.**

1. *If* $A \vdash_n S : T$, *then* $\mathcal{S}_n[\![A]\!] \vdash \mathcal{S}_n[\![S]\!] : \mathcal{S}_n[\![T]\!]$.
2. *If* $A \vdash_n C \triangleright T_1 \rightsquigarrow T_2$, *then* $\mathcal{S}_n[\![A]\!] \vdash \mathcal{C}_n[\![C]\!] : \mathcal{S}_n[\![T_1 \to T_2]\!]$.

Proof. By rule induction. ☐

Examples

Here are the translations of $\mathscr{C}_1$ and $\mathscr{C}_2$ from Section 2.

1. $\mathcal{C}_n[\![\mathscr{C}_1]\!] \equiv \mathcal{C}_n[\![?x.\mathtt{case}\, x\,\mathtt{of}\,(x_1, x_2) : x_1 < x_2]\!] \equiv$
   $\lambda z.\mathtt{let}\, y \Leftarrow z\,\mathtt{in}$
   $(\lambda x.\mathtt{let}\, x_S \Leftarrow \mathcal{S}_n[\![\mathtt{case}\, x\,\mathtt{of}\,(x_1, x_2) : x_1 < x_2]\!]\,\mathtt{in}$
   $\mathsf{if}\, x_S\,\mathsf{then}\, x\,\mathsf{else}\,\mathtt{raise}_{\mathsf{b}}\,)\,[y]$
2. $\mathcal{C}_n[\![\mathscr{C}_2]\!] \equiv \mathcal{C}_n[\![H^x_\to\,(?x.x \geq 0)\,(?y.y * y = x)]\!] \equiv$
   $\lambda z.\mathtt{let}\, z' \Leftarrow z\,\mathtt{in}\,[\lambda x.\mathcal{C}_n[\![?y.y * y = x]\!]\,(z'\,(\mathcal{C}_n[\![?x.x \geq 0]\!]\, x))]$

3.3 Meaning Preservation

To prove meaning preservation for typed terms, we define an alternative translation $\mathcal{S}'_n[\![]\!]$ that ignores contracts and construct a reduction correspondence between evaluations under $\mathcal{S}_n[\![]\!]$ and $\mathcal{S}'_n[\![]\!]$. In the following, $\Sigma, e \mapsto \Sigma', e'$ denotes the single-step reduction relation of $\lambda^{ML+\times}$, where Σ ranges over stores mapping locations to expressions. Moreover, $\Sigma, e \precsim' \Sigma', e'$ denotes syntactic equality up to the difference between terms transformed by $\mathcal{S}_n[\![S]\!]$ and $\mathcal{S}'_n[\![S]\!]$.

Theorem 2. *Suppose that S is a closed source term with $A \vdash_n S : T$ and that $\emptyset, \mathcal{S}_n[\![S]\!] \overset{*}{\mapsto} \Sigma, a$ is a finite reduction sequence with either $a \equiv [e]$ or $a \equiv \mathtt{raise}_\mathtt{u}$.*

Then there is a finite reduction sequence $\emptyset, \mathcal{S}'_n[\![S]\!] \overset{*}{\mapsto} \Sigma', a'$ with $\Sigma', a' \precsim' \Sigma, a$.

Because our effect system prevents nonterminating contract predicates we can prove that nontermination is not caused by contract monitoring.

Theorem 3. *If S is a closed source term with $A \vdash_n S : T$ such that there is an infinite $\mapsto$-reduction sequence starting with $\emptyset, \mathcal{S}_n[\![S]\!]$ then there is an infinite $\mapsto$-reduction sequence starting with $\emptyset, \mathcal{S}'_n[\![S]\!]$.*

3.4 Conjunction and Commutativity

The final property that we are after is idempotency. This property is surprisingly subtle and first requires some discussion about conjunction and commutativity of contracts. We then prove idempotency in Section 3.5.

Hinze et al. [12] observe that contracts are not idempotent in a lazy setting with non-termination as the only effect. It turns out that two factors contribute to the lack of idempotency for their contract language. First, their language includes a conjunction operator for contracts, which is interpreted as in $\lambda^{?!}$ as composition of contracts. Second, their counterexample for idempotency forms the conjunction of a predicate contract and a structural contract (*i.e.*, a function, pair, or sum contract).

The counterexample given by Hinze et al. is very similar to the example contract $\mathscr{C}_5$ defined in Section 2.2. We have already seen that $\mathscr{C}_5$ is not idempotent because

$$\mathtt{case}\ !\mathscr{C}_5\,(1,2)\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

evaluates to false whereas

$$\mathtt{case}\ !\mathscr{C}_5\,(!\mathscr{C}_5\,(1,2))\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

signals a contract violation.

The principal reason for this failure lies in the non-commutativity of $\mathscr{C}_1$ and $\mathscr{C}'_3$.

Definition 4 (Commutativity)**.** *Two contracts C_1, C_2* commute *if and only if, for all expressions S,*

$$!C_1\,(!C_2\,S) = !C_2\,(!C_1\,S)$$

where $=$ denotes equality in the metalanguage.

It is easy to see that $\mathscr{C}_1$ and $\mathscr{C}'_3$ do not commute. The expression

$$\mathscr{S}_1 \equiv \mathtt{case}\ !\mathscr{C}'_3\,(!\mathscr{C}_1\,(1,2))\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

evaluates to false whereas

$$\mathscr{S}_2 \equiv \mathtt{case}\ !\mathscr{C}_1\,(!\mathscr{C}'_3\,(1,2))\ \mathtt{of}\ (x_1,x_2) : \mathsf{false}$$

signals a contract violation.

We can show that the loss of commutativity (and hence idempotency) in the call-by-name setting is directly attributable to mixing a structural contract with a predicate contract that scrutinizes its argument. (A predicate contract scrutinizes its argument either through a case expression or by calling it as a function.)

Lemma 5 (Commutativity)**.**

1. *Predicate contracts commute with each other.*
2. *A predicate contract and a structural contract commute if the predicate contract does not scrutinize its argument.*
3. *Two structural contracts $C_1 \equiv H_\delta\,C'_1\,C''_1$ and $C_2 \equiv H_\delta\,C'_2\,C''_2$ commute if C'_1 commutes with C'_2 and C''_1 commutes with C''_2 (for $\delta \in \{\times, +, \rightarrow\}$).*
4. *A contract $C_1 \equiv \&\,C_{11}\,C_{12}$ and a contract C_2 commute if C_{11} and C_{12} both commute with C_2.*

Proof. By translation and equational reasoning in the metalanguage. □

It is tempting to ask whether we really need predicate contracts **and** structural contracts. For example, without structural contracts, commutativity would hold trivially. The following two propositions demonstrate that we lose expressiveness if we dispense with predicate or structural contracts.

Proposition 6. *The predicate contract $C \equiv ?x.\mathsf{false}$ cannot be simulated by any structural contract.*

Proof. By definition, $!C\,S$ raises an exception for any expression S. But for any of $C' \equiv H_\times\,C_1\,C_2$, $H_+\,C_1\,C_2$, or $H^y_\rightarrow\,C_1\,C_2$, there is a choice of S such that $!C'\,S$ yields a value (*i.e.*, does not raise an exception). For example, if $C' \equiv H_\times\,C_1\,C_2$ then choose $S \equiv (S_1, S_2)$ to obtain the result $!C'\,S = (!C_1\,S_1, !C_2\,S_2)$. □

Proposition 7. *The structural contract $C \equiv H_\times\,C'\,C'$ where $C' \equiv ?x.\mathsf{false}$ cannot be simulated by any predicate contract.*

Proof. It holds that $!C\,(1,2) = (!C'\,1, !C'\,2)$ and

$$\mathtt{case}\ !C\,(1,2)\ \mathtt{of}\ (x_1,x_2) : x_1 = !C'\,1 = \mathtt{raise}_\mathtt{b}$$

Let $C'' \equiv ?x.S$ be any predicate contract. If C'' is trivial (*i.e.*, S returns true for all x without scrutiny), then

$$\mathtt{case}\ !C''\,(1,2)\ \mathtt{of}\ (x_1,x_2) : x_1 = 1 \neq \mathtt{raise}_\mathtt{b}$$

If C'' is non-trivial, then there is some (S_1, S_2) for which C'' fails. But then the context

$$\mathtt{case}\ ![\,]\,(S_1,S_2)\ \mathtt{of}\ (x_1,x_2) : \mathsf{true}$$

distinguishes between C'' (result: fail) and C (result: true). □

3.5 Idempotency

Closer examination of the example in the preceding subsection shows that while $\mathscr{S}_1$ is well-typed, $\mathscr{S}_2$ is not. The source of the typing problem is $!\mathscr{C}_1\,(!\mathscr{C}'_3\,(1,2))$, which attempts to inject a blame effect inside the argument of $\mathscr{C}_1$. This observation motivates the typing rule for contract conjunction in Figure 2, which restricts conjunctions to commuting contracts, as shown by the following lemma.

Lemma 8. *If $A \vdash_n \&\,C_1\,C_2 \triangleright T \rightsquigarrow T'$ and $A \vdash_n S : T$ then $\mathcal{S}_n[\![!C_1\,(!C_2\,S)]\!] = \mathcal{S}_n[\![!C_2\,(!C_1\,S)]\!]$.*

Proof. From $A \vdash_n \&\,C_1\,C_2 \triangleright T \rightsquigarrow T'$ we obtain that C_1 and C_2 commute by performing a simultaneous induction on C_1 and C_2 and by applying Lemma 5 where appropriate. □

We now can prove idempotency.

Theorem 9. *Suppose that $A \vdash_n !C\,S : T$.*
Then $\mathcal{S}_n[\![!C\,(!C\,S)]\!] = \mathcal{S}_n[\![!C\,S]\!]$.

Proof. We proceed by induction on C. In each case, the claim follows by translation and equational reasoning in the metalanguage. □

4. Delayed Contract Monitoring

This section develops a novel alternative to eager contract monitoring for call-by-name. The motivation comes from recalling two examples in Section 2 and their typings in Section 3.1:

- The type of contract

  $$\mathscr{C}_1 \equiv ?x.\mathtt{case}\, x\, \mathtt{of}\, (x_1, x_2) : x_1 < x_2$$

  is

  $$(\mathtt{int}^{\epsilon_1} \times \mathtt{int}^{\epsilon_2})^{\epsilon'} \rightsquigarrow (\mathtt{int}^{\epsilon_1} \times \mathtt{int}^{\epsilon_2})^{\epsilon_1 \cup \epsilon_2 \cup \epsilon' \cup \mathsf{b}}$$

  with $\epsilon_1 \cup \epsilon_2 \subseteq \cup_\rho \mathsf{r}(\rho)$. Hence, we cannot attach it to the pair $(S, 0)$ for a possibly diverging S because $(S, 0)$ would have type $(\mathtt{int}^{\perp} \times \mathtt{int}^{\emptyset})^{\emptyset}$.
- Contract

  $$\mathscr{C}_2 \equiv H^x_{\rightarrow}\, (?x.x \geq 0)\, (?y.y * y = x)$$

  has type

  $$(\mathtt{int}^{\epsilon \cup \mathsf{b}} \to \mathtt{int}^{\epsilon'})^{\epsilon''} \rightsquigarrow (\mathtt{int}^{\epsilon} \to \mathtt{int}^{\epsilon \cup \epsilon' \cup \mathsf{b}})^{\epsilon''}$$

  with $\epsilon \subseteq \cup_\rho \mathsf{r}(\rho)$, so that a function of the form $!\mathscr{C}_2\, f$ cannot take an argument of type $\mathtt{int}^{\perp}$.

The problem with both example contracts is that they are too eager: $\mathscr{C}_1$ forces evaluation of both pair components, and $\mathscr{C}_2$ forces evaluation of the function argument. The type system restricts the effects of the subjects of $\mathscr{C}_1$ and $\mathscr{C}_2$ to guarantee meaning preservation and idempotency.

Delayed contract monitoring addresses this problem by staying in line with the program's observation of the expression under the call-by-name evaluation strategy. It only enforces observable contracts, in the sense that assertions are only checked if and when the user code observes a combination of values that violates the contract.

Delayed contract monitoring does not evaluate the predicate of $\mathscr{C}_1$ before the program has forced evaluation of both pair components. Similarly, the postcondition of $\mathscr{C}_2$ would only be checked if the function attached to $\mathscr{C}_2$ was strict.

There is also a logical point of view for comparing eager and delayed contract monitoring for functions. Let p and q be the pre- and postcondition of a function contract. The contract states that $p \Rightarrow q$ if the function terminates. By first testing the precondition, eager monitoring actually establishes a stronger condition, namely $p \wedge (p \Rightarrow q)$ which is equivalent to $p \wedge q$. It turns out that delayed monitoring establishes the same condition, but interprets it in a strict three valued logic with truth values **T**, **F**, and **U** (unknown) where the unknown outcome indicates that not all values needed are available, yet.

4.1 Intuitive Explanation

To enable delayed contract monitoring, contracts do no longer access subjects directly but through special reference cells called *boxes*. A box is empty as long as the corresponding part of the subject has not been reduced to a value, otherwise it carries the value. If contract enforcement finds a box empty, it delays evaluation of the contract until this box is filled.

Basically, delayed monitoring wraps the subject of a contract such that a box is associated with every inspected part of the value. This box is initially empty and it gets filled when the corresponding part is evaluated. Syntactically, *box variables* ξ access boxes.

For example, suppose S is a pair of type $\mathtt{int} \times \mathtt{int}$. If we now enforce a predicate contract on S, then delayed monitoring creates three boxes, a top-level box for the whole value and two subsidiary boxes for the `int` components. Once evaluation has produced a value (S_1, S_2), it fills the top-level box with this pair. If further evaluation produces values for S_1 or S_2, then these values are put into the appropriate subsidiary boxes.

The actual contract code is linked to these boxes in a subject-observer style. Initially, all assertions are put in an assertion list. Every time a new box gets filled, the system attempts to evaluate the assertions in the list. There are three possible outcomes. First, the assertion may succeed, in which case it is removed from the list. Second, the assertion may not be able to complete because it depends on boxes which are still empty. In this case, it is put back on the assertion list. Third, the assertion may fail, in which case the blame exception is raised.

To enable this sort of delayed contract monitoring, predicate contracts are restricted to inspect the subject only with special elimination expressions, `CASE`, which we call *patterns*[4] (see P in Figure 4). They always decompose a value bound to a box variable and bind its components to other box variables.

Examples

1. The predicate contract

   $$\mathscr{C}_1' \equiv ?\xi.\mathtt{CASE}\, \xi\, \mathtt{OF}\, (\xi_1, \xi_2) : \xi_1 < \xi_2$$

   asserts that the first component of a pair is smaller than the second, but only if the program forces both components of the pair. It corresponds to $\mathscr{C}_1$ in Section 2.2.
2. The function contract

   $$\mathscr{C}_2' \equiv H^{\xi'}_{\rightarrow}\, (?\xi.\xi \geq 0)\, (?\xi.\xi^2 = \xi')$$

   asserts that a function computes the square root when applied to an non-negative argument. Delayed contract monitoring enforces the pre- and postconditions of $\mathscr{C}_2'$ only if the subject of the contract is a strict function. $\mathscr{C}_2'$ corresponds to $\mathscr{C}_2$ in Section 2.2.
3. The predicate contract

   $$\begin{aligned} \mathscr{C}_7' \equiv ?\xi.\mathtt{CASE}\, \xi\, \mathtt{OF}\, (\xi_1, \xi_2) :& \\ \mathtt{CASE}\, \xi_1\, \mathtt{OF}\, (\xi_{11}, \_) :& \\ \mathtt{CASE}\, \xi_2\, \mathtt{OF}\, (\xi_{21}, \_) :&\ \xi_{11} = \xi_{21} \end{aligned}$$

   is a contract which asserts that $S_{11} = S_{21}$ holds for a pair of pairs $((S_{11}, S_{12}), (S_{21}, S_{22}))$. It is enforced as soon as the program demands both S_{11} and S_{21}.

The next subsection 4.2 formalizes a static semantics for delayed contract monitoring. Section 4.3 defines the dynamic semantics, again by translation to $\lambda^{ML+\times}$. Finally, Section 4.4 presents a Haskell implementation of delayed contract monitoring. For brevity, the formalization does not deal with sum types: their treatment is analogous to product types.

4.2 Static Semantics

Figure 4 shows the call-by-name source language with delayed monitoring. It is based on the syntax of expressions (Figure 1) as well as on the syntax of effects, types, environments, and the typing rules defined in Figure 2.

Each contract enforcement, $!_\rho C\, S$, carries a unique region annotation ρ specifying that only boxes allocated in ρ may be read in C. The user regions as well as regions for other contract enforcements are different from ρ. The expression form ξ denotes a box variable drawn from a countably infinite subset of box variables *BVar* $\subseteq$ *Var*. This special kind of variable is used to access the content of a box and it is only bound by contracts.

[4] Not to be mistaken for Haskell-like patterns. However, Template-Haskell[18] could serve to translate Haskell patterns into our pattern language.

Syntax; $\xi \in \mathit{BVar} \subseteq \mathit{Var}$

$$
\begin{aligned}
S &::= \dots \text{(as in Figure 1 but without } !C\,S) \cdots \mid\, !_\rho C\,S \mid \xi \\
C &::= ?\xi.P \mid H^{\xi}_{\rightarrow}\,C\,C \mid H_{\times}\,C\,C \mid \&\,C\,C \\
\mathit{Pat} \ni P &::= S \mid \mathtt{CASE}\,\xi\,\mathtt{OF}\ (\xi,\xi) : P \\
\mathsf{e} &::= \dots \text{(as in Figure 2)} \cdots \mid \mathsf{d} \\
U, T, A &::= \dots \text{(as in Figure 2)} \dots
\end{aligned}
$$

Typing rules for expression (as in Figure 2 but with two new rules)

$$
\frac{A(\xi) = \mathsf{box}_\rho(U^{\epsilon})^{\epsilon'}}{A \vdash \xi : U^{\epsilon \cup \epsilon' \cup \mathsf{r}(\rho) \cup \mathsf{d}}}
\qquad
\frac{A \vdash C \triangleright_\rho T \leadsto T' \quad A \vdash S : T}{A \vdash\, !_\rho C\,S : T'}
$$

Typing rules for contracts

$$
\frac{\begin{array}{c} A, \xi : \mathsf{box}_\rho(U_1^{\emptyset})^{\emptyset} \vdash P \triangleright_\rho \mathsf{bool}^{\epsilon} \\ \epsilon \subseteq \mathsf{r}(\rho) \cup \mathsf{d} \qquad U_1 \Downarrow^{\mathsf{r}(\rho)\cup\mathsf{w}(\rho)\cup\mathsf{b}} \mathcal{B}_\xi[\![P]\!] \mathrel{-\!\!\triangleright} U_2 \end{array}}{A \vdash ?\xi.P \triangleright_\rho U_1^{\epsilon'} \leadsto U_2^{\epsilon' \cup \mathsf{r}(\rho)\cup\mathsf{w}(\rho)\cup\mathsf{b}}}
$$

$$
\frac{A \vdash C_1 \triangleright_\rho U^{\epsilon'} \leadsto T_1 \quad A, \xi : \mathsf{box}_\rho(U^{\emptyset})^{\emptyset} \vdash C_2 \triangleright_\rho T_2 \leadsto T_2'}{A \vdash H^{\xi}_{\rightarrow}\,C_1\,C_2 \triangleright_\rho (T_1 \rightarrow T_2)^{\epsilon} \leadsto (U^{\epsilon'} \rightarrow T_2')^{\epsilon\cup\mathsf{r}(\rho)\cup\mathsf{w}(\rho)\cup\mathsf{b}}}
$$

$$
\frac{A \vdash C_1 \triangleright_\rho T_1 \leadsto T_1' \quad A \vdash C_2 \triangleright_\rho T_2 \leadsto T_2'}{A \vdash H_{\times}\,C_1\,C_2 \triangleright_\rho (T_1 \times T_2)^{\epsilon} \leadsto (T_1' \times T_2')^{\epsilon\cup\mathsf{r}(\rho)\cup\mathsf{w}(\rho)\cup\mathsf{b}}}
$$

Typing rules for patterns

$$
\frac{A \vdash S : T}{A \vdash S \triangleright_\rho T}
$$

$$
\frac{\begin{array}{c} A \vdash \xi : (U_1^{\epsilon_1} \times U_2^{\epsilon_2})^{\epsilon} \\ A, \xi_1 : \mathsf{box}_\rho(U_1^{\emptyset})^{\emptyset}, \xi_2 : \mathsf{box}_\rho(U_2^{\emptyset})^{\emptyset} \vdash P \triangleright_\rho U^{\epsilon} \end{array}}{A \vdash \mathtt{CASE}\,\xi\,\mathtt{OF}\ (\xi_1, \xi_2) : P \triangleright_\rho U^{\epsilon}}
$$

Figure 4: Source language for call-by-name with delayed monitoring.

Two contract forms are different from the forms in Figure 1: the function contract $H^{\xi}_{\rightarrow}\,C_1\,C_2$ makes the function argument available to the postcondition C_2 through the box ξ. The predicate contract $?\xi.P$ binds a box variable ξ and defines its predicate through a pattern P, which is either an expression or decomposes a box variables into a pair of box variables.

There is a new exception effect d which signals that the value of an empty box is accessed. The programmer cannot catch this exception, but contract monitoring uses it as a notification that a contract must be temporarily suspended.

The typing rules rely on the type abbreviations in Figure 6. A box-type $\mathsf{box}_\rho(t)$ is essentially a reference cell holding an optional value of type t. The typing rule for box variables indicates that accessing a box variable may find an empty box and thus raise a d exception. The rule for contract enforcement passes the region annotation into the typing judgment for contracts, $A \vdash C \triangleright_\rho T \leadsto T$.

The typing rule for predicate contracts allows the effect $\mathsf{r}(\rho) \cup \mathsf{d}$ in pattern P, so that P can read boxes in region ρ. [5] Moreover, the outgoing type U_2 has additional effects ($\mathsf{r}(\rho)$, $\mathsf{w}(\rho)$, and b) not only at its top-level but also at its subterms. This is achieved through the premise $U_1 \Downarrow^{\mathsf{r}(\rho)\cup\mathsf{w}(\rho)\cup\mathsf{b}} \mathcal{B}_\xi[\![P]\!] \mathrel{-\!\!\triangleright} U_2$, as explained in the next paragraph. Similarly, the rules for function and pair contracts add these effects to the top-level of their outgoing types. The additional effects are required because the dynamic semantics (see Figure 7) instruments the outgoing expressions to allow for delayed contract monitoring. The typing judgment for patterns, $A \vdash P \triangleright_\rho T$, asserts that P has type T.

The function $\mathcal{B}_\xi[\![P]\!]$ translates a pattern P into an annotated type B. The judgment $U \Downarrow^{\epsilon} B \mathrel{-\!\!\triangleright} U'$ uses the annotated type B to attach effect ϵ at certain positions of U, resulting in U'. Both are defined in Figure 5.

For example, consider contract $\mathscr{C}_7'$. To implement delayed contract monitoring, we have to translate this contract into a function that instruments a pair $((S_{11}, S_{12}), (S_{21}, S_{22}))$ such that the predicate $S_{11} = S_{21}$ is checked when both S_{11} and S_{21} have been reduced to values. But how do we perform this instrumentation? And, how does it change the type?

An annotated type B mirrors the structure of a type U and specifies at which positions inside U instrumentations and type changes are necessary. For an annotated type $\top$, no instrumentation and no change of type is necessary. The annotated type $\overline{\xi_1}@B_1 \times \overline{\xi_2}@B_2$ specifies that a pair (S_1, S_2) has to be instrumented such that once S_1 (or S_2) becomes a value, all boxes ξ_{1_i} (or ξ_{2_i}) are filled with this value and delayed contracts are enforced. The type changes accordingly. (The notation $\overline{\xi}$ used in this figure denotes the sequence $\xi_1, \dots, \xi_n$, and $\cdot$ stands for the empty sequence.)

The function $\mathcal{B}_\xi[\![P]\!]$ translates a pattern P with respect to box variable ξ into an annotated type. Its definition exploits that patterns scrutinize only box variables. The operator $\oplus$ merges box annotations.

As an example, suppose that P_7' is the pattern used in $\mathscr{C}_7'$. After eliminating wildcards _ from the sequences of box variables, we have

$$
\mathcal{B}_\xi[\![P_7']\!] = \xi_1@(\xi_{11}@\top \times \cdot@\top) \times \xi_2@(\xi_{21}@\top \times \cdot@\top)
$$

The judgment $U \Downarrow^{\epsilon} B \mathrel{-\!\!\triangleright} U'$ adds the effect ϵ to those positions in U where the corresponding position in B has a non-empty sequence of box annotations. The resulting type is U'. For

Syntax

$$
\mathit{BType} \ni B ::= \top \mid \overline{\xi}@B \times \overline{\xi}@B
$$

Translation from patterns into annotated types

$$
\begin{aligned}
\mathcal{B}_\xi[\![S]\!] &= \top \\
\mathcal{B}_\xi[\![\mathtt{CASE}\,\xi\,\mathtt{OF}\ (\xi', \xi'') : P]\!] &= \mathcal{B}_\xi[\![P]\!] \oplus \\
&\qquad (\xi'@\mathcal{B}_{\xi'}[\![P]\!] \times \xi''@\mathcal{B}_{\xi''}[\![P]\!]) \\
\mathcal{B}_{\xi_0}[\![\mathtt{CASE}\,\xi\,\mathtt{OF}\ (\xi', \xi'') : P]\!] &= \mathcal{B}_{\xi_0}[\![P]\!] \qquad \textbf{if } \xi \neq \xi_0
\end{aligned}
$$

Merging of box annotations

$$
\top \oplus B = B \qquad B \oplus \top = B
$$

$$
\begin{array}{c}
(\overline{\xi_1}@B_1 \times \overline{\xi_2}@B_2) \oplus (\overline{\xi_1'}@B_1' \times \overline{\xi_2'}@B_2') = \\
\overline{\xi_1\xi_1'}@(B_1 \oplus B_1') \times \overline{\xi_2\xi_2'}@(B_2 \oplus B_2')
\end{array}
$$

Effect enrichment based on annotated types

$$
U \Downarrow^{\epsilon} \top \mathrel{-\!\!\triangleright} U
$$

$$
\frac{U_i \Downarrow^{\epsilon} B_i \mathrel{-\!\!\triangleright} U_{i+2} \quad \epsilon_{i+2} = \mathrm{Eff}(\overline{\xi_i}, \epsilon) \quad (\text{for } i = 1, 2)}{U_1^{\epsilon_1} \times U_2^{\epsilon_2} \Downarrow^{\epsilon} \overline{\xi_1}@B_1 \times \overline{\xi_2}@B_2 \mathrel{-\!\!\triangleright} U_3^{\epsilon_3} \times U_4^{\epsilon_4}}
$$

$$
\mathrm{Eff}(\cdot, \epsilon) = \emptyset \qquad\qquad \mathrm{Eff}(\overline{\xi}, \epsilon) = \epsilon \ \textbf{if } \overline{\xi} \neq \cdot
$$

Figure 5: Types with box annotations.

[5] Arbitrary read effects $\cup_\rho \mathsf{r}(\rho)$ are no longer allowed in the predicate expression because delayed contract monitoring would cause them to be propagated to the conclusion, regardless of whether they actually appear in the predicate. It is not difficult to adapt the system presented in this section to allow arbitrary read effects. We regard this choice as bad design: the results of those reads are unpredictable because assertions may be evaluated at any time.

example,

$$\begin{array}{l}
\quad (\mathtt{int}^{\epsilon_{11}} \times \mathtt{int}^{\epsilon_{12}})^{\epsilon_1} \times (\mathtt{int}^{\epsilon_{21}} \times \mathtt{int}^{\epsilon_{22}})^{\epsilon_2} \\
\Downarrow^{\epsilon'} \xi_1 @ (\xi_{11} @ \top \times \cdot @ \top) \times \xi_2 @ (\xi_{21} @ \top \times \cdot @ \top) \\
\multimap (\mathtt{int}^{\epsilon_{11} \cup \epsilon'} \times \mathtt{int}^{\epsilon_{12}})^{\epsilon_1 \cup \epsilon'} \times (\mathtt{int}^{\epsilon_{21} \cup \epsilon'} \times \mathtt{int}^{\epsilon_{22}})^{\epsilon_2 \cup \epsilon'}
\end{array}$$

Examples

We now can assign types to the examples from Section 4.1. In the following, $\epsilon' = \mathsf{r}(\rho) \cup \mathsf{w}(\rho) \cup \mathsf{b}$.

1. $A \vdash \mathscr{C}_1' \rhd_\rho (\mathtt{int}^{\epsilon_1} \times \mathtt{int}^{\epsilon_2})^{\epsilon} \rightsquigarrow (\mathtt{int}^{\epsilon_1 \cup \epsilon'} \times \mathtt{int}^{\epsilon_2 \cup \epsilon'})^{\epsilon \cup \epsilon'}$
2. $A \vdash \mathscr{C}_2' \rhd_\rho (\mathtt{int}^{\epsilon_1 \cup \epsilon'} \to \mathtt{int}^{\epsilon_1})^{\epsilon_2} \rightsquigarrow (\mathtt{int}^{\epsilon_1} \to \mathtt{int}^{\epsilon_1 \cup \epsilon'})^{\epsilon_2 \cup \epsilon'}$
3. $A \vdash \mathscr{C}_7' \rhd ((\mathtt{int}^{\epsilon_{11}} \times \mathtt{int}^{\epsilon_{12}})^{\epsilon_1} \times (\mathtt{int}^{\epsilon_{21}} \times \mathtt{int}^{\epsilon_{22}})^{\epsilon_2})^{\epsilon} \rightsquigarrow ((\mathtt{int}^{\epsilon_{11} \cup \epsilon'} \times \mathtt{int}^{\epsilon_{12}})^{\epsilon_1 \cup \epsilon'} \times (\mathtt{int}^{\epsilon_{21} \cup \epsilon'} \times \mathtt{int}^{\epsilon_{22}})^{\epsilon_2 \cup \epsilon'})^{\epsilon \cup \epsilon'}$

The typings show that there are **no** restrictions on effects. Hence, delayed contract monitoring solves the problem discussed at the beginning of this section.

4.3 Dynamic Semantics

The translation into the monadic metalanguage makes use of the definitions in Figure 6. The type cstore_ρ is the type of contract stores, that is, references to delayed contract expressions. The expression $\mathsf{fill\text{-}box}$ puts a value into an optional box if the box is empty and not none. The expression $\mathsf{demand\text{-}box}$ reads a value from a box and throws the d exception if the box is empty. The $\mathsf{add\text{-}to\text{-}cstore}$ expression extends a contract store with a contract expression e of type $\mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{d}\cup\mathsf{b}}(*)$ after masking out its d effect. The expression $\mathsf{check\text{-}cstore}$ evaluates the contract expression in the given contract store.

Figure 7 defines the translation into the monadic metalanguage. We treat box variables ξ as ordinary variables of the metalanguage. The translation of contracts, $\mathcal{C}_l[\![C]\!]\rho\, w\, e$, is now parameterized: ρ is the region for boxes and the contract store, variable w represents the contract store, and e is either none or $\mathsf{some}\, e'$ where e' is a box intended to hold the value being checked by C.

The translation of a predicate contract $?\xi.P$ is a function that takes the subject and returns an instrumented version of it. The function body allocates boxes corresponding to ξ and the boxes bound in P, adds a suitable checking expression to the contract store, and enforces delayed contracts. The set $\mathrm{Boxes}(P)$ consists of all boxes bound by a `CASE` expression in P. The expression added to the contract store w is constructed using the auxiliary translation function $\mathcal{P}[\![]\!]$ defined in Figure 8: $\mathcal{P}[\![P]\!]$ navigates through the `CASE` structure of P to find the predicate to evaluate. Another auxiliary function of Figure 8, $\mathcal{I}[\![]\!]$, performs the instrumentation by inserting the appropriate $\mathsf{fill\text{-}box}$ and $\mathsf{check\text{-}cstore}$ operations.

The translation of a function contract yields a function that first allocates a new box and then behaves like the composition of the postcondition, the real function, and the precondition. The last argument to the translation of the precondition is $\mathsf{some}\, x'$, which causes the box to be filled as soon as the function consumes its argument.

Figure 9 shows the translation of example $\mathscr{C}_1'$. The new translation in Figure 7 is also compatible with typing:

Theorem 10 (Translation Preserves Typing)**.**
1. *If $A \vdash S : T$, then $\mathcal{S}_l[\![A]\!] \vdash \mathcal{S}_l[\![S]\!] : \mathcal{S}_l[\![T]\!]$.*
2. *If $A \vdash C \rhd_\rho U^\epsilon \rightsquigarrow T$ and $\Gamma \supseteq \mathcal{S}_l[\![A]\!]$ with $\Gamma(x) = \mathsf{cstore}_\rho$ and $\Gamma \vdash e : \mathsf{option}(\mathsf{box}_\rho(\mathbf{T}_\emptyset(\mathcal{S}_l[\![U]\!])))$, then $\Gamma \vdash \mathcal{C}_l[\![C]\!]\rho\, x\, e : \mathcal{S}_l[\![U^\epsilon \to T]\!]$.*

Proof. We strengthen the proposition by adding suitable cases for the translation functions $\mathcal{P}, \mathcal{I}$, and $\mathcal{I}'$, and proceed by simultaneous induction. □

$$\begin{array}{l}
\mathcal{C}_l[\![\mathscr{C}_1']\!]\rho\, w\, e \equiv \mathcal{C}_l[\![?\xi.\mathtt{CASE}\,\xi\,\mathtt{OF}\,(\xi_1, \xi_2) : \xi_1 < \xi_2]\!]\rho\, w\, e \equiv \\
\quad \lambda z.\mathtt{let}\, y \Leftarrow z\, \mathtt{in}\, \mathtt{let}\, x \Leftarrow \mathsf{new\text{-}box}_\rho\, \mathtt{in} \\
\qquad \mathtt{let}\, x_1 \Leftarrow \mathsf{new\text{-}box}_\rho\, \mathtt{in}\, \mathtt{let}\, x_2 \Leftarrow \mathsf{new\text{-}box}_\rho\, \mathtt{in} \\
\qquad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, e\, [y]\, \mathtt{in}\, \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, (\mathsf{some}\, x)\, [y]\, \mathtt{in} \\
\qquad \mathtt{let}\, _ \Leftarrow \mathsf{add\text{-}to\text{-}cstore}\, w\, ((\lambda\xi.\lambda\xi_1.\lambda\xi_2. \\
\qquad\quad \mathtt{let}\, x_S \Leftarrow (\mathtt{let}\, _ \Leftarrow \mathcal{S}_l[\![\xi]\!]\, \mathtt{in}\, \mathcal{S}_l[\![\xi_1 < \xi_2]\!])\, \mathtt{in} \\
\qquad\qquad \mathsf{if}\, x_S\, \mathsf{then}\, [()]\, \mathsf{else}\, \mathtt{raise}_{\mathsf{b}})\, [x]\, [x_1]\, [x_2])\, \mathtt{in} \\
\qquad \mathtt{let}\, _ \Leftarrow \mathsf{check\text{-}cstore}\, w\, \mathtt{in} \\
\qquad\quad \mathtt{case}\, y\, \mathsf{of}\, (y_1, y_2) : [(\mathtt{let}\, x' \Leftarrow y_1\, \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, (\mathsf{some}\, x_1)\, [x']\, \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{let}\, _ \Leftarrow \mathsf{check\text{-}cstore}\, w\, \mathtt{in}\, [x'] \\
\qquad\qquad\qquad\qquad\quad , \\
\qquad\qquad\qquad\qquad\qquad \mathtt{let}\, x' \Leftarrow y_2\, \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, (\mathsf{some}\, x_2)\, [x']\, \mathtt{in} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{let}\, _ \Leftarrow \mathsf{check\text{-}cstore}\, w\, \mathtt{in}\, [x'])]
\end{array}$$

Figure 9: Example $\mathscr{C}_1'$ translated to the monadic metalanguage.

```
data Contract :: * -> * where
  Prop     :: (Box a -> P a) -> Contract a
  Function :: Contract a -> (Box a -> Contract b)
           -> Contract (a -> b)

data P :: * -> * where
  CasePair :: Box (a,b)
           -> ((Box a, Box b) -> P c) -> P (a,b)
  Pred     :: CM Bool -> P a

data Box a   -- abstract
data CM a    -- abstract, instance of Monad

demand :: Box a -> CM a
assert :: Contract a -> a -> a
```

Figure 10: Interface of a Haskell library for delayed contract monitoring.

Delayed monitoring preserves the meaning by construction. Potential failure of MP for eager monitoring is caused by predicate contracts that evaluate the subject beyond weak head normal form. Delayed monitoring by definition does not perform any evaluation beyond that of the user code.

Idempotency holds for essentially the same reason. Both observations rely crucially on the fact that the effects of the user code affect different memory regions than each of the contract enforcements. Essentially, a side-effect during contract-enforcement does not affect the outcome of the user code at all, except for possibly causing a contract violation.

4.4 Implementation

We have developed a Haskell implementation of delayed contract monitoring that is based on the interface of Hinze et al. [12]. It is available from `http://proglang.informatik.uni-freiburg.de/projects/contracts/`. Our implementation does not enforce the effect system used in the formalization. It is the responsibility of the programmer to prevent side-effects from invalidating the meaning preservation or idempotency properties. We have tested our code under the Glasgow Haskell Compiler [10].

Figure 10 shows the interface of our library.[6] Following Hinze et al. we use Generalized Algebraic Data Types (GADTs) to model contracts in Haskell. As in their implementation, `Contract a` is the type of contracts for values of type `a`. Following our formalization,

[6] We simplified the interface for the sake of presentation. The real implementation supports blame assignment and additional contract forms.

Type abbreviations

$$\begin{aligned}
\mathsf{option}(t) &\equiv t + \mathbf{T}_\emptyset(*)\\
\mathsf{box}_\rho(t) &\equiv \mathtt{ref}_\rho\, \mathbf{T}_\emptyset(\mathsf{option}(t))\\
\mathsf{cstore}_\rho &\equiv \mathtt{ref}_\rho\, \mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{b}}(*)
\end{aligned}$$

Expression abbreviations

$$\begin{aligned}
\mathsf{some} &: t \to \mathsf{option}(t) \equiv \lambda x.\mathtt{in}_1\, x\\
\mathsf{none} &: \mathsf{option}(t) \equiv \mathtt{in}_2\, [()]\\
\mathsf{new\text{-}box}_\rho &: \mathbf{T}_\emptyset(\mathsf{box}_\rho(t)) \equiv \mathtt{ref}_\rho\, [\mathsf{none}]\\
\mathsf{fill\text{-}box} &: \mathsf{option}(\mathsf{box}_\rho(t)) \to t \to \mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{w}(\rho)}(*) \equiv \lambda x.\lambda y.\mathtt{case}\, x\, \mathtt{of}\\
&\qquad \mathtt{in}_1\, x' \colon \mathtt{let}\, z \Leftarrow \mathtt{get}\, x'\, \mathtt{in}\, \mathtt{let}\, z' \Leftarrow z\, \mathtt{in}\\
&\qquad\qquad \mathtt{case}\, z'\, \mathtt{of}\, \mathtt{in}_1\, _ : [()]\\
&\qquad\qquad\qquad \mathtt{in}_2\, _ : \mathtt{set}\, x'\, [\mathsf{some}\, y]\\
&\qquad \mathtt{in}_2\, _ \, : [()]\\
\mathsf{demand\text{-}box} &: \mathsf{box}_\rho(t) \to \mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{d}}(t) \equiv \lambda x.\mathtt{let}\, y \Leftarrow \mathtt{get}\, x\, \mathtt{in}\, \mathtt{let}\, y' \Leftarrow y\, \mathtt{in}\\
&\qquad \mathtt{case}\, y'\, \mathtt{of}\, \mathtt{in}_1\, z : [z]\\
&\qquad\qquad \mathtt{in}_2\, _ : \mathtt{raise}_{\mathsf{d}}\\
\mathsf{new\text{-}cstore}\rho &: \mathbf{T}_\emptyset(\mathsf{cstore}_\rho) \equiv \mathtt{ref}_\rho\, [()]\\
\mathsf{add\text{-}to\text{-}cstore} &: \mathsf{cstore}_\rho \to \mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{d}\cup\mathsf{b}}(*) \to \mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{w}(\rho)}(*) \equiv \lambda x.\lambda y.\mathtt{let}\, z \Leftarrow \mathtt{get}\, x\, \mathtt{in}\\
&\qquad \mathtt{set}\, x\, (\mathtt{let}\, _ \Leftarrow \mathtt{try}\, y\, \mathtt{catch}_{\mathsf{d}}.[()]\, \mathtt{in}\, z)\\
\mathsf{check\text{-}cstore} &: \mathsf{cstore}_\rho \to \mathbf{T}_{\mathsf{r}(\rho)\cup\mathsf{b}}(*) \equiv \lambda x.\mathtt{let}\, y \Leftarrow \mathtt{get}\, x\, \mathtt{in}\, y
\end{aligned}$$

Figure 6: Abbreviations for the monadic metalanguage.

$$\begin{aligned}
\mathcal{S}_l[\![\xi]\!] &\equiv \mathtt{let}\, y \Leftarrow \xi\, \mathtt{in}\, \mathtt{let}\, x \Leftarrow \mathsf{demand\text{-}box}\, y\, \mathtt{in}\, x\\
\mathcal{S}_l[\![S]\!] &\equiv \mathcal{S}_n[\![S]\!] \qquad \text{if } S \neq \xi\\
\\
\mathcal{S}_l[\![!_\rho C\, S]\!] &\equiv \mathtt{let}\, x \Leftarrow \mathsf{new\text{-}cstore}\, \mathtt{in}\, (\mathcal{C}_l[\![C]\!]\rho\, x\, \mathsf{none})\, \mathcal{S}_l[\![S]\!]\\
\mathcal{C}_l[\![?\xi_0.P]\!]\rho\, w\, e &\equiv \lambda z.\mathtt{let}\, y \Leftarrow z\, \mathtt{in}\\
&\quad \mathtt{let}\, x_0 \Leftarrow \mathsf{new\text{-}box}_\rho\, \mathtt{in}\, \ldots \mathtt{let}\, x_n \Leftarrow \mathsf{new\text{-}box}_\rho\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, e\, [y]\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, (\mathsf{some}\, x_0)\, [y]\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{add\text{-}to\text{-}cstore}\, w\, ((\lambda\xi_0.\ldots\lambda\xi_n.\mathtt{let}\, x_S \Leftarrow \mathcal{P}[\![P]\!]\, \mathtt{in}\, \mathsf{if}\, x_S\, \mathsf{then}\, [()]\, \mathsf{else}\, \mathtt{raise}_{\mathsf{b}}\,)\, [x_0]\ldots[x_n])\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{check\text{-}cstore}\, w\, \mathtt{in}\\
&\qquad ((\lambda\xi_0.\ldots\lambda\xi_n.\mathcal{I}[\![\mathcal{B}_\xi[\![P]\!]]\!]w)\, (\mathsf{some}\, x_0)\ldots(\mathsf{some}\, x_n))\, y\\
&\quad \textbf{where } \mathrm{Boxes}(P) = \{\xi_1,\ldots,\xi_n\}\\
\mathcal{C}_l[\![H^{\xi}_{\to}\, C_1\, C_2]\!]\rho\, w\, e &\equiv \lambda z.\mathtt{let}\, y \Leftarrow z\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, e\, [y]\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{check\text{-}cstore}\, w\, \mathtt{in}\\
&\qquad [\lambda x.\mathtt{let}\, x' \Leftarrow \mathsf{new\text{-}box}_\rho\, \mathtt{in}\, (((\lambda\xi.\mathcal{C}_l[\![C_2]\!]\rho\, w\, \mathsf{none})\, [x']) \circ y \circ \mathcal{C}_l[\![C_1]\!]\rho\, w\, (\mathsf{some}\, x'))\, x]\\
\mathcal{C}_l[\![H_\times\, C_1\, C_2]\!]\rho\, w\, e &\equiv \lambda z.\mathtt{let}\, y \Leftarrow z\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{fill\text{-}box}\, e\, [y]\, \mathtt{in}\\
&\quad \mathtt{let}\, _ \Leftarrow \mathsf{check\text{-}cstore}\, w\, \mathtt{in}\\
&\qquad \mathsf{case}\, y\, \mathsf{of}\, (x_1, x_2) : [(\mathcal{C}_l[\![C_1]\!]\rho\, w\, \mathsf{none}\, x_1, \mathcal{C}_l[\![C_2]\!]\rho\, w\, \mathsf{none}\, x_2)]\\
\mathcal{C}_l[\![\&\, C_1\, C_2]\!]\rho\, w\, e &\equiv \lambda z.\mathcal{C}_l[\![C_1]\!]\rho\, w\, e\, (\mathcal{C}_l[\![C_2]\!]\rho\, w\, e\, z)\\
\\
\mathcal{S}_l[\![U]\!] \equiv \mathcal{S}_n[\![U]\!] &\qquad \mathcal{S}_l[\![T]\!] \equiv \mathcal{S}_n[\![T]\!] \qquad \mathcal{S}_l[\![A]\!] \equiv \mathcal{S}_n[\![A]\!]
\end{aligned}$$

Figure 7: Translation of call-by-name with delayed contract monitoring.

Predicate extraction; $\mathcal{P}[\![]\!]$: *Pat* $\rightarrow$ *Exp*

$$\mathcal{P}[\![S]\!] \equiv \mathcal{S}_l[\![S]\!]$$
$$\mathcal{P}[\![\mathtt{CASE}\,\xi\,\mathtt{OF}\,(\xi_1,\xi_2) : P]\!] \equiv \mathtt{let}\,\_ \Leftarrow \mathcal{S}_l[\![\xi]\!]\,\mathtt{in}\,\mathcal{P}[\![P]\!]$$

Instrumentation with fill-box operations; $\mathcal{I}[\![]\!]$: *BType* $\rightarrow$ *Exp* $\rightarrow$ *Exp*, $\quad \mathcal{I}'$: *Exp* $\rightarrow$ *Exp* $\rightarrow$ $\overline{BVar}$ $\rightarrow$ *BType* $\rightarrow$ *Exp*

$$\mathcal{I}[\![\top]\!]w \equiv \lambda x.[x]$$
$$\mathcal{I}[\![\overline{\xi_1}@B_1 \times \overline{\xi_2}@B_2]\!]w \equiv \lambda y.\mathtt{case}\,y\,\mathtt{of}\,(y_1,y_2) : [(\mathcal{I}'\,y_1\,w\,\overline{\xi_1}\,B_1, \mathcal{I}'\,y_2\,w\,\overline{\xi_2}\,B_2)]$$
$$\begin{aligned}\mathcal{I}'\,e\,w\,(\xi_1\ldots\xi_n)\,B \equiv\ & \mathtt{let}\,x \Leftarrow e\,\mathtt{in} \\ & \mathtt{let}\,\_ \Leftarrow \textsf{fill-box}\ \xi_1\,[x]\,\mathtt{in}\ \ldots\ \mathtt{let}\,\_ \Leftarrow \textsf{fill-box}\ \xi_n\,[x]\,\mathtt{in} \\ & \mathtt{let}\,\_ \Leftarrow \textsf{check-cstore}\ w\,\mathtt{in} \qquad \textbf{omit if}\ n < 1 \\ & (\mathcal{I}[\![B]\!]w)\,x\end{aligned}$$

Figure 8: Auxiliary functions for the translation of call-by-name with delayed contract monitoring.

a predicate contract `Prop` is constructed from a function of type `Box a -> P a`.

A pattern of type `P a` scrutinizes values of type `a`. The `Pred` constructor takes a value of the monadic type `CM Bool` because predicates typically use the effectful function `demand` to read box values. The `demand` function is the only operation in the `CM` monad.

The following code is the Haskell version of the $\mathscr{C}'_7$ contract:

```
example :: Contract ((Int,a), (Int,b))
example = Prop (\abcd ->
            CasePair abcd (\(ab,cd) ->
              CasePair ab (\(a,b) ->
                CasePair cd (\(c,d) ->
                  Pred (do x <- demand a
                           y <- demand c
                           return $ x == y)))))
```

The `assert` function enforces contracts. This interactive GHCi session demonstrates that our implementation delays contract monitoring until all relevant values are available.

```
*Main> let p = assert example ((1,2),(3,4))
*Main> fst p
(1,2)
*Main> snd (snd p)
4
*Main> fst (snd p)
*** Exception: contract violation
```

5. Related Work

Findler and Felleisen [7] give a semantic account of contract monitoring in an object-oriented setting. Most of the difficulties with contracts in object-oriented languages arise from subtyping, which our languages do not have. Their proof of contract soundness informally assumes that pre- and postconditions of contracts are free of effects. In contrast, our work employs an effect system that statically rules out programs with effectful contract expressions.

A follow-up article [8] provides a contract system for higher-order functions. Their contract calculus λ^{CON} extends the λ-calculus with contracts, similar to our language $\lambda^{?!}$. In addition to the features of $\lambda^{?!}$, their calculus λ^{CON} includes first-class contracts and blame assignment. These extensions should be orthogonal to the aspects discussed in this paper. The dynamic semantics of λ^{CON} is call-by-value, whereas we define the dynamic semantics of $\lambda^{?!}$ for call-by-value and call-by-name. Moreover, contracts in $\lambda^{?!}$ preserve the meaning of programs and are idempotent, a result which has not been formally verified for λ^{CON}.

Blume and McAllester [3] prove soundness and completeness for a system with higher-order contracts. Their system is inspired by previous work of Findler and Felleisen [8]. However, their system only considers non-termination and the exceptions introduced by contract violations as effects. Our monadic contract system supports also effects like mutable state and is further extensible.

Findler and Blume [6] model contracts by pairs of domain projections. They provide an ordering relation on contracts and give further insights on the most permissive contract. In addition, they address the problem of proper blame assignment. Like our work, their model is also driven by the central properties of meaning preservation and idempotency.

Chitil et al. [5] argue that an assertion should not force extra evaluation in a lazy language like Haskell and define and implement a language of lazy assertions. No formal properties have been established for lazy assertions. We come to a similar conclusion for semantic reasons and thus develop formal underpinnings for their work, too. Our implementation of lazy monitoring is novel and significantly different from theirs, for instance, they rely on threads to evaluate contracts. Hence, their system defers evaluation of assertions until the assertion checking thread runs, whereas our implementation guarantees that contracts are enforced as soon as all values involved are available.

Inspired by this work, Chitil and Huch [4] define a pattern logic for specifying lazy assertions. Their pattern language can express richer assertions than our system with delayed contract monitoring. For example, they are able to express assertions on recursively defined data structures. However, their work lacks a formal basis and they do not prove meaning preservation or idempotency for their assertions.

Hinze et al. [12] introduce a DSL for contracts in Haskell. Our implementation of delayed contract monitoring extends their DSL, although it includes features which are not supported by our formalization: first-class contracts, blame assignment, and nonterminating contract predicates. We expect the first two features to be orthogonal extensions of our formal system. Termination, however, is required to prove meaning preservation.

Wadler [21] shows that effect systems are closely related to effect-indexed monads. Because contracts themselves involve an effect, namely raising a blame exception, it is not surprising that contracts can be realized with monads. In a sense, we introduce polygamy by also "marrying" monads with contracts.

Besides the work of Benton and Kennedy [2] that investigates the equational theory of effect-indexed monads in the context of the computational lambda calculus λ_C, Führmann's work [9] considers various sublanguages of λ_C which are axiomatized by equations. Führmann considers notions like "repeatable", "central", and "discardable" which are closely related to our equations *(repeat)*, *(swap)*, and *(zero)*. However, instead of linking the validity of equations to the presence or absence of certain effects, he establishes all containment relations between the sublanguages.

6. Conclusion and Future Work

We define type and effect systems for impure languages with call-by-name evaluation and contracts. Contract monitoring comes in two variants: eager monitoring and delayed monitoring. A translation into a monadic metalanguage defines the semantics of each variant. The type and effect systems enforce restrictions on contracts that enable the proof that erasing contracts does not change the meaning of programs and that contract monitoring is idempotent. An interesting outcome of the formalization is that eager monitoring is either too restrictive or destroys desirable properties of contract monitoring for call-by-name languages. Thus delayed monitoring is the appropriate mechanism for call-by-name. We demonstrate that delayed monitoring is practical by providing a novel implementation in Haskell.

Although delayed monitoring is superior to eager monitoring, it is not the silver bullet for contract monitoring in call-by-name languages. Consider the example

$$\mathsf{launch} \equiv !(?c.c == 0)\ (\lambda c.\mathsf{start\text{-}rocket}\ ())$$

which implements a function for launching a hypothetical mars-lander mission. The argument to launch is the current countdown value. Clearly, launch should only be called when the countdown is zero, so we use the contract $?c.c == 0$ to install this requirement as a precondition. But with both eager and delayed monitoring, preconditions are only checked when the function demands its argument. As $\lambda c.\mathsf{start\text{-}rocket}\ ()$ never demands its argument, the rocket lifts off, no matter what the current countdown value is!

The example above may seem artificial but there are also more realistic ones. Consider the following Haskell function that searches a sorted list for a specific element.

```
search :: [Int] -> Int -> Bool
search []     _              = False
search (x:xs) y | x == y     = True
                | x > y      = False
                | otherwise = search xs y
```

We now would like to express the requirement “the input list is sorted” as a precondition of this function. However, if the function checks the precondition lazily, then `search [1,102,42] 42` would not report a contract violation (because the function demands only the sorted prefix `1,102` of the list) and returns `False`.

The question is now: Is there a contract monitoring strategy for call-by-name languages that has the benefits of delayed monitoring **and** prohibits examples such as the ones given? An answer to this question remains for future work.

References

[1] P. Abercrombie and M. Karaorman. jContractor: Design by contract for Java. http://jcontractor.sourceforge.net/, 2003.

[2] N. Benton and A. Kennedy. Monads, effects, and transformations. In *The Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, Sept. 1999. Elsevier.

[3] M. Blume and D. McAllester. A sound (and complete) model of contracts. In K. Fisher, editor, *Proc. ICFP 2004*, pages 189–200, Snowbird, Utah, USA, Sept. 2004. ACM Press, New York.

[4] O. Chitil and F. Huch. A pattern logic for prompt lazy assertions. In *Implementation and Application of Functional Languages, 18th International Workshop, IFL 2006*, number 4449 in LNCS, April 2007.

[5] O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In P. Trinder, G. Michaelson, and R. Pena, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, number 3145 in LNCS. Springer, November 2004.

[6] R. B. Findler and M. Blume. Contracts as pairs of projections. In *Proc. Eighth International Symposium on Functional and Logic Programming FLOPS 2006*, Fuji Susono, Japan, Apr. 2006. Springer.

[7] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. 16th ACM Conf. OOPSLA*, pages 1–15, Tampa Bay, FL, USA, 2001. ACM Press, New York.

[8] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In S. Peyton-Jones, editor, *Proc. ICFP 2002*, pages 48–59, Pittsburgh, PA, USA, Oct. 2002. ACM Press, New York.

[9] C. Führmann. Varieties of effects. In M. Nielsen, editor, *Foundations of Software Science and Computation Structures, FOSSACS 2002*, number 2303 in LNCS, pages 144–158, Grenoble, France, Apr. 2002. Springer.

[10] GHC. The Glasgow Haskell compiler. http://www.haskell.org/ghc/, 2008.

[11] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[12] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Proc. Eighth International Symposium on Functional and Logic Programming FLOPS 2006*, Fuji Susono, Japan, Apr. 2006. Springer.

[13] R. Kramer. iContract — the Java design by contract tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295– 307, Los Alamitos, CA, USA, 1998.

[14] B. Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, Oct. 1992.

[15] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[16] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[17] E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.

[18] T. Sheard and S. P. Jones. Template meta-programming for haskell. In M. Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop*, Pittsburgh, PA, USA, 2002.

[19] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *J. Funct. Program.*, 2(3):245–272, July 1992.

[20] The PLT Group. *PLT MzLib: Libraries Manual*. Rice University, University of Utah, Dec. 2005. Version 300.

[21] P. Wadler and P. Thiemann. The marriage of monads and effects. *ACM Trans. Computational Logic*, 4(1):1–32, Jan. 2003.

A. The Extended Monadic Metalanguage

The monadic metalanguage λ^{ML} [17] is the term language for strong computational monads. The present work uses $\lambda^{ML+\times}$, an extension of λ^{ML} with sums and products inspired by MIL [2], as the target language for transformations that define the dynamic semantics of $\lambda^{?!}$ under eager and delayed contract monitoring. The general setup extends Wadler's [21] with sum and product types and considers a monad with nontermination, mutable state, and exceptions.

Figure 11 defines syntax, type language, effectful constants, notion of reduction, and evaluation contexts of $\lambda^{ML+\times}$, relying on the definition of effects in Figure 2. The notation $\mathtt{let}\, x \Leftarrow e\, \mathtt{in}\, e'$ stands for the monad's bind operation and $[e]$ for the return operation. The effectful operations are available through type-indexed families of constants. The constant $\mathtt{rec}$ is indexed by $t_1 \to \mathbf{T}_\epsilon(t_2)$, $\mathtt{raise}_{\mathsf{e}}$ and $\mathtt{try\_catch}_{\mathsf{e}}$ are indexed by $\mathbf{T}_{\mathsf{e}}(t)$, and the operations on references are indexed by t.

The judgment $\Gamma \vdash e : t$ assigns type t to expression e using standard typing rules [2, 21].

A *store* is a partial mapping $\Sigma : \mathit{Loc} \to \mathit{Exp}$ where $\mathit{Loc} \subseteq \mathit{Var}$ is a countably infinite set of locations. Locations are not bound by function expressions. We let ℓ range over locations, $\Sigma[\ell \mapsto e]$ updates store Σ at $\ell \in \mathrm{Dom}(\Sigma)$ and $\Sigma \uplus \{\ell \mapsto e\}$ extends Σ at $\ell \notin \mathrm{Dom}(\Sigma)$.

A *configuration* is a pair Σ, e of a store and an expression. Evaluation of $\lambda^{ML+\times}$ is defined through a deterministic small-step reduction relation $\mapsto$, where $\Sigma_1, e_1 \mapsto \Sigma_2, e_2$ if either (a) $e_1 \equiv \mathcal{M}[e_1']$ and $\Sigma_1, e_1' \longrightarrow \Sigma_2, e_2'$ and $e_2 \equiv \mathcal{M}[e_2']$, or (b) $e_1 \equiv \mathcal{M}[\mathcal{E}[e_1']]$ and $e_1' \longrightarrow e_2'$ and $\Sigma_1 = \Sigma_2$ and $e_2 \equiv \mathcal{M}[\mathcal{E}[e_2']]$.

Theorem 11 (Type soundness)**.** *The type system of $\lambda^{ML+\times}$ is sound.*

Many proofs in this paper rely on $\lambda^{ML+\times}$'s equational theory. Its equality relation $=$ is the compatible closure of the equation schemes in Figure 12. The equalities *(beta-…)* are derived from the reductions. The equalities *(eta-let)* and *(let-let)* are derived from the monadic axioms [17]. The remaining *(let-…)* equalities are commuting conversions.

The last three rules derive type- and effect-dependent equalities. Rule *(repeat)* defines repeatable computations which can be evaluated twice in a row without producing different results. They must not perform both read and write operations in the same region. Rule *(swap)* states that exchangeable computations can be performed in any order. The side conditions on the rule can be relaxed, but they are sufficient for our purposes. Rule *(zero)* describes when raising an exception can be considered as a zero in a monad.

Benton and Kennedy [2] prove that the rules *(beta-…)*, *(eta-let)*, as well as *(let-…)* are operational equivalences, and that the rules *(repeat)* and *(swap)* are equivalences in MIL. Similar techniques apply to the proof for rule *(zero)*.

Expressions, values, answers, and types; $x \in \mathit{Var}$, $c \in \mathit{Const}$

$$\begin{array}{rl}
\mathit{Exp} \ni e ::= & c \mid x \mid \lambda x.e \mid e\,e \mid () \mid \mathtt{case}\,e\,\mathtt{of}\,() : e \mid (e,e)\mathtt{case}\,e\,\mathtt{of}\,(x,x) : e \mid \mathtt{in}_i\,e \\
& \mid \mathtt{case}\,e\,\mathtt{of}\,\mathtt{in}_1\,x : e; \mathtt{in}_2\,x : e\mathtt{let}\,x \Leftarrow e\,\mathtt{in}\,e \mid [e] \\
\mathit{Value} \ni v ::= & c \mid \ell \mid \lambda x.e \mid () \mid (v,v) \mid \mathtt{in}_i\,v\mathtt{rec}\,e \mid \mathtt{try}\,a\,\mathtt{catch}_{\mathsf{e}} \mid \mathtt{set}\,v \mid [e] \\
\mathit{Answer} \ni a ::= & v \mid \mathtt{raise}_{\mathsf{e}} \\
\mathit{Type} \ni t ::= & * \mid \mathtt{ref}_\rho\,t \mid t \times t \mid t + t \mid t \to t \mid \mathbf{T}_\epsilon(t)
\end{array}$$

Type schemes for effectful operations

$$\begin{array}{rl}
\mathtt{rec} : & \forall t_1, t_1.((t_1 \to \mathbf{T}_{\epsilon \cup \bot}(t_2)) \to (t_1 \to \mathbf{T}_\epsilon(t_2))) \to t_1 \to \mathbf{T}_\epsilon(t_2) \\
\mathtt{raise}_{\mathsf{e}} : & \forall t.\mathbf{T}_{\mathsf{e}}(t) \\
\mathtt{try_catch}_{\mathsf{e}} : & \forall t.\mathbf{T}_{\epsilon_1}(t) \to \mathbf{T}_{\epsilon_2}(t) \to \mathbf{T}_{(\epsilon_1 \setminus \mathsf{e}) \cup \epsilon_2}(t) \\
\mathtt{ref}_\rho : & \forall t.t \to \mathbf{T}_\emptyset(\mathtt{ref}_\rho\,t) \\
\mathtt{get} : & \forall t.\mathtt{ref}_\rho\,t \to \mathbf{T}_{\mathsf{r}(\rho)}(t) \\
\mathtt{set} : & \forall t.\mathtt{ref}_\rho\,t \to t \to \mathbf{T}_{\mathsf{w}(\rho)}(*)
\end{array}$$

Notion of reduction

$$\begin{array}{ll}
(\lambda x.e_1)\,e_2 & \longrightarrow e_1[x \mapsto e_2] \\
\mathtt{case}\,()\,\mathtt{of}\,() : e & \longrightarrow e \\
\mathtt{case}\,(e_1,e_2)\,\mathtt{of}\,(x_1,x_2) : e & \longrightarrow e[x_1 \mapsto e_1, x_2 \mapsto e_2] \\
\mathtt{case}\,\mathtt{in}_i\,e\,\mathtt{of}\,\mathtt{in}_1\,x_1 : e_1; & \longrightarrow e_i[x_i \mapsto e] \\
\qquad\qquad\quad \mathtt{in}_2\,x_2 : e_2 & \\
\mathtt{let}\,x \Leftarrow [e_1]\,\mathtt{in}\,e_2 & \longrightarrow e_2[x \mapsto e_1] \\
\Sigma, \mathtt{rec}\,(\lambda f.e_1)\,e_2 & \longrightarrow \Sigma, (\lambda f.e_1)\,(\mathtt{rec}\,(\lambda f.e_1))\,e_2 \\
\Sigma, \mathtt{let}\,x \Leftarrow \mathtt{raise}_{\mathsf{e}}\,\mathtt{in}\,e & \longrightarrow \Sigma, \mathtt{raise}_{\mathsf{e}} \\
\Sigma, \mathtt{try}\,\mathtt{raise}_{\mathsf{e}}\,\mathtt{catch}_{\mathsf{e}}.e & \longrightarrow \Sigma, e \\
\Sigma, \mathtt{try}\,\mathtt{raise}_{\mathsf{e}}\,\mathtt{catch}_{\mathsf{e}'}.e & \longrightarrow \Sigma, \mathtt{raise}_{\mathsf{e}} \quad \textbf{if}\ \mathsf{e}' \neq \mathsf{e} \\
\Sigma, \mathtt{try}\,[e]\,\mathtt{catch}_{\mathsf{e}}.e' & \longrightarrow \Sigma, [e] \\
\Sigma, \mathtt{ref}_\rho\,e & \longrightarrow \Sigma \uplus \{\ell \mapsto e\}, [\ell] \\
\Sigma, \mathtt{get}\,\ell & \longrightarrow \Sigma, [\Sigma(\ell)] \\
\Sigma, \mathtt{set}\,\ell\,e & \longrightarrow \Sigma[\ell \mapsto e], [()]
\end{array}$$

Evaluation contexts

$$\begin{array}{rl}
\mathcal{E} ::= & \square \mid \mathcal{E}\,e \mid \mathtt{case}\,\mathcal{E}\,\mathtt{of}\,() : e \mid \mathtt{case}\,\mathcal{E}\,\mathtt{of}\,(x,x) : e\mathtt{case}\,\mathcal{E}\,\mathtt{of}\,\mathtt{in}_1\,x : e; \mathtt{in}_2\,x : e \mid \mathtt{rec}\,\mathcal{E} \mid \mathtt{get}\,\mathcal{E} \mid \mathtt{set}\,\mathcal{E} \\
\mathcal{M} ::= & \square \mid \mathtt{let}\,x \Leftarrow \mathcal{M}\,\mathtt{in}\,e \mid \mathtt{try}\,\mathcal{M}\,\mathtt{catch}_{\mathsf{e}}.e
\end{array}$$

Figure 11: Extended monadic metalanguage.

$$\begin{array}{lllll}
\textit{(beta)} & (\lambda x.e_1)\,e_2 & = & e_1[x \mapsto e_2] \\
\textit{(beta-unit)} & \mathtt{case}\,()\,\mathtt{of}\,() : e & = & e \\
\textit{(beta-pair)} & \mathtt{case}\,(e_1,e_2)\,\mathtt{of}\,(x_1,x_2) : e & = & e[x_1 \mapsto e_1, x_2 \mapsto e_2] \\
\textit{(beta-sum)} & \mathtt{case}\,\mathtt{in}_i\,e\,\mathtt{of}\,\mathtt{in}_1\,x_1 : e_1; \mathtt{in}_2\,x_2 : e_2 & = & e_i[x_i \mapsto e] \\
\textit{(beta-let)} & \mathtt{let}\,x \Leftarrow [e_1]\,\mathtt{in}\,e_2 & = & e_2[x \mapsto e_1] \\
\textit{(eta-let)} & \mathtt{let}\,x \Leftarrow e\,\mathtt{in}\,[x] & = & e \\
\textit{(let-let)} & \mathtt{let}\,x \Leftarrow (\mathtt{let}\,y \Leftarrow e_1\,\mathtt{in}\,e_2)\,\mathtt{in}\,e_3 & = & \mathtt{let}\,y \Leftarrow e_1\,\mathtt{in}\,(\mathtt{let}\,x \Leftarrow e_2\,\mathtt{in}\,e_3) \\
\textit{(let-sum)} & \mathtt{let}\,x \Leftarrow (\mathtt{case}\,e\,\mathtt{of}\,\mathtt{in}_i\,x_i : e_i)\,\mathtt{in}\,e' & = & \mathtt{case}\,e\,\mathtt{of}\,\mathtt{in}_i\,x_i : \mathtt{let}\,x \Leftarrow e_i\,\mathtt{in}\,e' \\
\textit{(let-pair)} & \mathtt{let}\,x \Leftarrow (\mathtt{case}\,e\,\mathtt{of}\,(x_1,x_2) : e')\,\mathtt{in}\,e'' & = & \mathtt{case}\,e\,\mathtt{of}\,(x_1,x_2) : \mathtt{let}\,x \Leftarrow e'\,\mathtt{in}\,e'' \\
\textit{(let-unit)} & \mathtt{let}\,x \Leftarrow (\mathtt{case}\,e\,\mathtt{of}\,() : e')\,\mathtt{in}\,e'' & = & \mathtt{case}\,e\,\mathtt{of}\,() : \mathtt{let}\,x \Leftarrow e'\,\mathtt{in}\,e''
\end{array}$$

$$\textit{(repeat)}\ \frac{\Gamma \vdash e_1 : \mathbf{T}_{\epsilon_1}(t_1) \quad \Gamma, x : t_1, y : t_1 \vdash e_2 : \mathbf{T}_{\epsilon_2}(t_2) \quad (\forall \rho)\ \mathsf{w}(\rho) \cup \mathsf{r}(\rho) \not\subseteq \epsilon_1}{\Gamma \vdash \quad \mathtt{let}\,x \Leftarrow e_1\,\mathtt{in}\,\mathtt{let}\,y \Leftarrow e_1\,\mathtt{in}\,e_2 = \mathtt{let}\,x \Leftarrow e_1\,\mathtt{in}\,\mathtt{let}\,y \Leftarrow [x]\,\mathtt{in}\,e_2 : \mathbf{T}_{\epsilon_1 \cup \epsilon_2}(t_2)}$$

$$\textit{(swap)}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathbf{T}_{\epsilon_1}(t_1) \quad \Gamma \vdash e_2 : \mathbf{T}_{\epsilon_2}(t_2) \\ \Gamma, x_1 : t_1, x_2 : t_2 \vdash e_3 : \mathbf{T}_{\epsilon_3}(t_3) \quad \epsilon_1, \epsilon_2 \subseteq \bot \cup \cup_\rho \mathsf{r}(\rho)\end{array}}{\begin{array}{rl}\Gamma \vdash & \mathtt{let}\,x_1 \Leftarrow e_1\,\mathtt{in}\,\mathtt{let}\,x_2 \Leftarrow e_2\,\mathtt{in}\,e_3 \\ = & \mathtt{let}\,x_2 \Leftarrow e_2\,\mathtt{in}\,\mathtt{let}\,x_1 \Leftarrow e_1\,\mathtt{in}\,e_3 : \mathbf{T}_{\epsilon_1 \cup \epsilon_2 \cup \epsilon_3}(t_3)\end{array}} \qquad \textit{(zero)}\ \frac{\Gamma \vdash e_1 : \mathbf{T}_{\mathsf{e} \cup \cup_\rho \mathsf{r}(\rho)}(t_1)}{\Gamma \vdash \mathtt{let}\,x \Leftarrow e_1\,\mathtt{in}\,\mathtt{raise}_{\mathsf{e}} = \mathtt{raise}_{\mathsf{e}} : \mathbf{T}_{\mathsf{e}}(t)}$$

Figure 12: Equational theory of the metalanguage.