

# Semantics of Automatic Mutual Exclusion

Notes on definitions and proofs

Martín Abadi    Andrew Birrell    Tim Harris    Michael Isard

March 19, 2008

# Contents

<b>1</b>	<b>The Language</b>	<b>4</b>
<b>2</b>	<b>High-level, Strong Operational Semantics</b>	<b>5</b>
2.1	States . . . . .	5
2.2	Evaluation Model . . . . .	7
2.3	Steps . . . . .	7
<b>3</b>	<b>A Type System for Yielding</b>	<b>9</b>
3.1	The Type System . . . . .	9
3.2	Soundness of the Type System for Yielding . . . . .	11
3.2.1	Extending the Language with Assertions . . . . .	11
3.2.2	Typing States . . . . .	12
3.2.3	Auxiliary Results . . . . .	13
3.2.4	Computation Preserves Typability . . . . .	15
3.2.5	Progress . . . . .	17
<b>4</b>	<b>A Type System for Separation</b>	<b>20</b>
4.1	The Type System . . . . .	20
4.2	Soundness of the Type System for Separation . . . . .	22
4.2.1	Typing States . . . . .	23
4.2.2	Auxiliary Results . . . . .	23
4.2.3	Computation Preserves Typability . . . . .	25
4.2.4	Progress . . . . .	26
<b>5</b>	<b>A Weaker Semantics</b>	<b>27</b>
5.1	The Weak Semantics . . . . .	28
5.2	A Hypothesis . . . . .	29
5.3	Correctness . . . . .	30
<b>6</b>	<b>Roll-back</b>	<b>33</b>
6.1	An Informal Account . . . . .	34
6.2	A Lower-Level Semantics with Roll-back . . . . .	34
6.2.1	States . . . . .	34
6.2.2	Steps: Weak Semantics with Roll-back . . . . .	35
6.2.3	Steps: Strong Semantics with Roll-back . . . . .	38
6.3	Correctness . . . . .	38
6.3.1	Strong Semantics with Roll-back Implements Strong Semantics . . . . .	39

6.3.2	Weak Semantics with Roll-back Implements Strong Semantics with Roll-back . . . . .	41
6.3.3	Weak Semantics with Roll-back Implements Strong Semantics . . . . .	43
<b>7</b>	<b>Optimistic Concurrency</b>	<b>44</b>
7.1	An Informal Account . . . . .	44
7.2	A Lower-Level Semantics with Optimistic Concurrency . . . .	46
7.2.1	States . . . . .	46
7.2.2	Steps . . . . .	47
7.3	Soundness of the Type System for Separation, Revisited . . .	49
7.3.1	Typing States . . . . .	49
7.3.2	Auxiliary Results . . . . .	50
7.3.3	Computation Preserves Typability . . . . .	50
7.4	Correctness . . . . .	51

---

$V \in$	<i>Value</i>	=	$c \mid x \mid \lambda x. e$
$c \in$	<i>Const</i>	=	<code>unit</code> $\mid$ <code>false</code> $\mid$ <code>true</code>
$x, y \in$	<i>Var</i>		
$e, f \in$	<i>Exp</i>	=	$V$
			$e f$
			<code>ref</code> $e \mid !e \mid e := f$
			<code>async</code> $e$
			<code>blockUntil</code> $e$
			<code>unprotected</code> $e$

Figure 1: Syntax of the AME calculus.

---

## 1 The Language

The syntax of the language is defined in Figure 1. We call it the AME calculus, though undoubtedly other calculi with AME are possible.

Several other constructs are definable:

- Informally, we abbreviate  $(\lambda x. e') e$  to `let`  $x = e$  `in`  $e'$ .
- We also abbreviate `let`  $x = e$  `in`  $e'$  to  $e; e'$  when  $x$  does not occur free in  $e'$ .
- We treat `yield` as syntactic sugar for `unprotected unit`.
- We can express “abort and retry later” as `blockUntil false`.
- Traditional atomic blocks typically occur in the context of unprotected expressions, and differ from asynchronous blocks in that they are supposed to be executed immediately, not in some indefinite future. We can express `atomic`  $e$  as

```

let  $x =$  ref false in
  async ( $e$ ; unprotected ( $x :=$  true));
  blockUntil !x

```

where  $x$  is a fresh variable. The assignment to  $x$  is done without protection in order to avoid a violation of the kind discussed below.

For clarity, we may consider example programs using an extended language that uses these abbreviations and with other standard data types and notations.

---

**State space**

$$\begin{aligned} S &\in \text{State} = \text{RefStore} \times \text{ExpSeq} \times \text{Exp} \\ \sigma &\in \text{RefStore} = \text{RefLoc} \rightarrow \text{Value} \\ r &\in \text{RefLoc} \subset \text{Var} \\ T &\in \text{ExpSeq} = \text{Exp}^* \end{aligned}$$

Figure 2: State space.

---

**Evaluation contexts**

$$\begin{aligned} \mathcal{P} &= [] \mid \mathcal{P} e \mid V \mathcal{P} \mid \text{ref } \mathcal{P} \mid !\mathcal{P} \mid \mathcal{P} := e \mid r := \mathcal{P} \mid \text{blockUntil } \mathcal{P} \\ \mathcal{U} &= \text{unprotected } \mathcal{E} \mid \mathcal{U} e \mid V \mathcal{U} \mid \text{ref } \mathcal{U} \mid !\mathcal{U} \mid \mathcal{U} := e \mid r := \mathcal{U} \mid \text{blockUntil } \mathcal{U} \\ \mathcal{E} &= [] \mid \mathcal{E} e \mid V \mathcal{E} \mid \text{ref } \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \mid \text{blockUntil } \mathcal{E} \mid \text{unprotected } \mathcal{E} \\ \mathcal{F} &= T\mathcal{U}.T', \text{unit} \mid T, \mathcal{P} \end{aligned}$$

Figure 3: Evaluation contexts.

---

## 2 High-level, Strong Operational Semantics

This section describes a semantics for the language. It includes an evaluation model that is somewhat independent of the details of the language. This evaluation model is not entirely obvious, and (if properly explained) it could be of value as an informal specification even in the context of richer languages. Formally, the core of the operational semantics is a set of rules defined in Figures 2, 3, and 4.

### 2.1 States

As described in Figure 2, a state  $\langle \sigma, T, e \rangle$  consists of the following components:

- a reference store  $\sigma$ ,
- a collection of expressions  $T$  (each of which represents an asynchronous computation still to be performed), which we call the expression pool,
- a distinguished active expression  $e$ .

A reference store  $\sigma$  is a finite mapping of reference locations to values. Reference locations are simply special kinds of variables that can be bound

---

**Transition rules**

$\langle \sigma, \mathcal{F} [ (\lambda x. e) V ] \rangle$	$\mapsto_s \langle \sigma, \mathcal{F} [ e[V/x] ] \rangle$	(Trans Appl) <sub>s</sub>
$\langle \sigma, \mathcal{F} [ \text{ref } V ] \rangle$	$\mapsto_s \langle \sigma[r \mapsto V], \mathcal{F} [ r ] \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	(Trans Ref) <sub>s</sub>
$\langle \sigma, \mathcal{F} [ !r ] \rangle$	$\mapsto_s \langle \sigma, \mathcal{F} [ V ] \rangle$ if $\sigma(r) = V$	(Trans Deref) <sub>s</sub>
$\langle \sigma, \mathcal{F} [ r := V ] \rangle$	$\mapsto_s \langle \sigma[r \mapsto V], \mathcal{F} [ \text{unit} ] \rangle$	(Trans Set) <sub>s</sub>
$\langle \sigma, \mathcal{F} [ \text{async } e ] \rangle$	$\mapsto_s \langle \sigma, e. \mathcal{F} [ \text{unit} ] \rangle$	(Trans Async) <sub>s</sub>
$\langle \sigma, \mathcal{F} [ \text{blockUntil true} ] \rangle$	$\mapsto_s \langle \sigma, \mathcal{F} [ \text{unit} ] \rangle$	(Trans Block) <sub>s</sub>
$\langle \sigma, T, \mathcal{P} [ \text{unprotected } e ] \rangle$	$\mapsto_s \langle \sigma, T. \mathcal{P} [ \text{unprotected } e ], \text{unit} \rangle$	(Trans Unprotect) <sub>s</sub>
$\langle \sigma, T. \mathcal{E} [ \text{unprotected } V ]. T', \text{unit} \rangle$	$\mapsto_s \langle \sigma, T. \mathcal{E} [ V ]. T', \text{unit} \rangle$	(Trans Close) <sub>s</sub>
$\langle \sigma, T. e. T', \text{unit} \rangle$	$\mapsto_s \langle \sigma, T. T', e \rangle$	(Trans Activate) <sub>s</sub>

Figure 4: Transition rules of the abstract machine (strong).

---

only by the respective store. We write  $RefLoc$  for the set of reference locations. We assume that  $RefLoc$  is infinite, so  $RefLoc - dom(\sigma)$  is never empty.

We identify expressions with threads of computation. The semantics does not require the use of stacks.

For every state  $\langle \sigma, T, e \rangle$ , we require that if  $r \in RefLoc$  occurs free in  $\sigma(r')$ , in  $T$ , or in  $e$ , then  $r \in dom(\sigma)$ . This condition will be assumed for initial states and will be preserved by computation steps.

## 2.2 Evaluation Model

The evaluation of a program starts in an initial state  $\langle \sigma, e, \mathbf{unit} \rangle$  with a single thread and with  $\mathbf{unit}$  as distinguished active expression. In examples, the reference store  $\sigma$  may be assumed to have certain properties, for instance to be empty or to include specific distinguished locations for inputs and results of computations.

Evaluation then takes place according to the rules (given below) that specify the behavior of the various constructs in the language. The execution of threads is interleaved in a non-deterministic manner, subject to atomicity constraints.

Each evaluation step produces a new state. Given a state, the next state is determined by the next possible operation in the active expression or in one of the other expressions, but in the latter case only if those operations are part of a subexpression **unprotected**  $e$ .

The evaluation terminates when no further evaluation steps are possible. The evaluation may not terminate, and it may also get stuck, for instance with all threads blocked on some false condition.

The present semantics does not explicitly model optimistic concurrency schemes with “undo” facilities, nor backtracking from stuck states. An implementation may of course include such refinements. We consider some of them below.

## 2.3 Steps

In all cases, “the next possible operation” in an expression is found by decomposing the expression into an evaluation context and a subexpression that describes this operation.

As usual, a context is an expression with a hole  $[ \ ]$ , and an evaluation context is a context of a particular kind. Given a context  $\mathcal{C}$  and an expression  $e$ , we write  $\mathcal{C}[ e ]$  for the result of placing  $e$  in the hole in  $\mathcal{C}$ ; similarly, given

contexts  $\mathcal{C}$  and  $\mathcal{C}'$ , we write  $\mathcal{C}[\mathcal{C}']$  for the result of placing  $\mathcal{C}'$  in the hole in  $\mathcal{C}$ . We use several kinds of evaluation contexts:

- $\mathcal{P}$  evaluation contexts are for the execution of atomic fragments: the position for evaluation is not under **unprotected**.
- $\mathcal{U}$  evaluation contexts are for the execution of unprotected fragments: the position for evaluation is under **unprotected**.
- $\mathcal{E}$  evaluation contexts allow us to manipulate **unprotected** values in the execution of unprotected fragments.

We also let some evaluation contexts be sequences of expressions with a hole:

- $\mathcal{F}$  evaluation contexts are of the form  $T\mathcal{U}.T'$ , **unit** or of the form  $T, \mathcal{P}$ .

Thus,  $\mathcal{F}[e]$  is either of the form  $T\mathcal{U}[e].T'$ , **unit** or of the form  $T, \mathcal{P}[e]$ . We write  $e_0.\mathcal{F}[e_1]$  as an abbreviation for  $e_0.T\mathcal{U}[e_1].T'$ , **unit** or  $e_0.T, \mathcal{P}[e_1]$ , respectively.

These evaluation contexts are defined in Figure 3.

(In the cases of function application, the value  $V$  could be replaced with an abstraction. This would be a little more consistent with the treatment of operations on reference locations, perhaps, but makes no difference for our results.)

The abstract machine described in Figures 3 and 4 specifies the operational semantics of our language. Figure 4 gives rules that specify the transition relation that takes execution from one state to the next. The string “Trans” in the names of the rules refers to “transition” rules, not to “transactions”. In these rules, we write  $e[V/x]$  for the result of the capture-free substitution of  $V$  for  $x$  in  $e$ , and write  $\sigma[r \mapsto V]$  for the store that agrees with  $\sigma$  except at  $r$ , which is mapped to  $V$ . The subscript  $s$  in  $\mapsto_s$  indicates that this is a strong semantics, in the sense that it forbids interleavings of executions of atomic fragments and unprotected fragments.

For **yield**, if it were not syntactic sugar, we could have the extra rules:

$$\langle \sigma, T, \mathcal{P}[\text{yield}] \rangle \mapsto_s \langle \sigma, T, \mathcal{P}[\text{yield}], \text{unit} \rangle$$

and

$$\langle \sigma, T, \mathcal{E}[\text{yield}].T', e' \rangle \mapsto_s \langle \sigma, T, \mathcal{E}[\text{unit}].T', e' \rangle$$

---


$$\begin{array}{lcl}
s, t \in Type & = & \text{Unit} \\
& & | \text{Bool} \\
& & | s \rightarrow^p t \\
& & | \text{Ref } t \\
p, q \in & \{ & \text{Yields, NoYields} \}
\end{array}$$

Figure 5: Types for yielding.

---

### 3 A Type System for Yielding

We focus on the strong semantics. The results carry-over to the weak semantics, straightforwardly. Similar results may be obtained for lower-level semantics.

#### 3.1 The Type System

Figure 5 gives the syntax of types.

The type of an expression depends on a *typing environment*  $E$ , which maps variables to types. The typing environment is organized as a sequence of bindings, and we use  $\emptyset$  to denote the empty environment.

$$E ::= \emptyset \mid E, x : t$$

We define the type system using judgments. These judgments are described in Figure 6, together with rules for reasoning about the judgments. The core of the type system is the set of rules for the judgment  $E; p \vdash e : t$  (read “ $e$  is a well-typed expression of type  $t$  in typing environment  $E$  with effect  $p$ ”). The intent is that, if this judgment holds, then  $e$  yields values of type  $t$  with effect  $p$ , and the free variables of  $e$  are given bindings consistent with the typing environment  $E$ . When  $p$  is **Yields**, this means that the evaluation of  $e$  may yield; when  $p$  is **NoYields**, this means that the evaluation of  $e$  definitely does not yield. We write  $q <: p$  for  $p = q$  or  $p = \text{Yields}$ . We say that  $e$  is well-typed when there exist  $E$ ,  $p$ , and  $t$  such that  $E; p \vdash e : t$ .

The system is mostly straightforward. One noteworthy detail is that we require that the expression  $e$  in `async e` should be of type **Unit**; our results go through in slightly modified form without this condition.

One delicate design choice is that we arrange that every expression that can be typed with effect **NoYields** can also be typed with effect **Yields**. For instance, we allow giving the effect **Yields** to the constant `true`, although

---

**Judgments**

$E \vdash \diamond$        $E$  is a well-formed typing environment  
 $E; p \vdash e : t$        $e$  is a well-typed expression of type  $t$  in  $E$  with  $p$

**Rules**

$\emptyset \vdash \diamond$	(Env $\emptyset$ )
$\frac{E \vdash \diamond \quad x \notin \text{dom}(E)}{E, x : t \vdash \diamond}$	(Env $x$ )
$\frac{E \vdash \diamond}{E; p \vdash \mathbf{unit} : \mathbf{Unit}}$	(Exp Unit)
$\frac{E \vdash \diamond}{E; p \vdash \mathbf{false} : \mathbf{Bool}}$	(Exp Bool <b>false</b> )
$\frac{E \vdash \diamond}{E; p \vdash \mathbf{true} : \mathbf{Bool}}$	(Exp Bool <b>true</b> )
$\frac{E, x : t, E' \vdash \diamond}{E, x : t, E'; p \vdash x : t}$	(Exp $x$ )
$\frac{E, x : s; p \vdash e : t}{E; q \vdash \lambda x. e : s \rightarrow^p t}$	(Exp Fun)
$\frac{E; p \vdash e_1 : s \rightarrow^q t \quad E; p \vdash e_2 : s \quad q <: p}{E; p \vdash e_1 e_2 : t}$	(Exp Appl)
$\frac{E; p \vdash e : t}{E; p \vdash \mathbf{ref} e : \mathbf{Ref} t}$	(Exp Ref)
$\frac{E; p \vdash e : \mathbf{Ref} t}{E; p \vdash !e : t}$	(Exp Deref)
$\frac{E; p \vdash e_1 : \mathbf{Ref} t \quad E; p \vdash e_2 : t}{E; p \vdash e_1 := e_2 : \mathbf{Unit}}$	(Exp Set)
$\frac{E; p \vdash e : \mathbf{Unit}}{E; q \vdash \mathbf{async} e : \mathbf{Unit}}$	(Exp Async)
$\frac{E; p \vdash e : \mathbf{Bool}}{E; p \vdash \mathbf{blockUntil} e : \mathbf{Unit}}$	(Exp Block)
$\frac{E; p \vdash e : t}{E; \mathbf{yields} \vdash \mathbf{unprotected} e : t}$	(Exp Unprotect)

Figure 6: The first-order type system for yielding.

---

the evaluation of `true` will obviously never yield. This property ensures that effects are not invalidated by computation. For example, consider the expression `unprotected true`, which has effect `Yields` and produces the result `true`; because `true` has effects `NoYields` and also `Yields`, the effect of `unprotected true` continues to be derivable after reduction to `true`.

There are alternative approaches to achieving the same effect. These include the use of a system with subtyping, which would also provide more flexibility at function types. The present approach is simpler and enables us to focus on the core system. Undoubtedly richer type disciplines are possible.

## 3.2 Soundness of the Type System for Yielding

Initially the purpose of this proof was to force a close look at the definitions of the operational semantics and the type system. It served that purpose, and the definitions changed as a result. The proof is less straightforward than I expected.

### 3.2.1 Extending the Language with Assertions

We extend the calculus with a construct that asserts the absence of yields in a computation. This construct serves for expressing the correctness of the type system. (Intuitively, the correctness of the type system is the property that says that if an expression has effect `NoYields` statically then it does not yield at run-time. However, in the course of evaluation, the expression may change somewhat, and that should not be an excuse for yielding. So it is convenient to tag the expression, and to keep the tag on the expression even if the expression changes until its evaluation completes. An assertion serves as a tag.) This construct may also be useful for dynamic checking, conceivably, if the type system is not flexible enough.

The additions to the language are follows:

- We extend the syntax of the language with terms of the form  $\langle e \rangle$ . Informally,  $\langle e \rangle$  means that there is no yield in the course of the evaluation of  $e$ . (This notation is meant to be reminiscent of the angle brackets that indicate atomicity in the work of Lamport and others.)
- We extend the evaluation contexts of kinds  $\mathcal{P}$ ,  $\mathcal{U}$ , and  $\mathcal{E}$  respectively with  $\langle \mathcal{P} \rangle$ ,  $\langle \mathcal{U} \rangle$ , and  $\langle \mathcal{E} \rangle$  and we correspondingly extend the definition of  $\mathcal{F}$ .

- We add a rule to the operational semantics:

$$\langle \sigma, \mathcal{F}[ \langle V \rangle ] \rangle \mapsto_s \langle \sigma, \mathcal{F}[ V ] \rangle \quad (\text{Trans Assert})_s$$

Given that  $\langle e \rangle$  means that there is no yield in the course of the evaluation of  $e$ , this rule says that the assertion can be removed when  $e$  is a value  $V$  (not subject to further evaluation).

- For typing, we require that these assertions appear only attached to expressions with effect `NoYields`.

$$\frac{E; \text{NoYields} \vdash e : t}{E; p \vdash \langle e \rangle : t} \quad (\text{Exp Assert})$$

This extension is conservative, in the sense that it does not affect the operational semantics or the typing of expressions without assertions. Therefore, the main results below (Theorems 3.7 and 3.11) apply also without this extension.

Consider a transition that is an instance of  $(\text{Trans Unprotect})_s$ , so this transition is of the form

$$\langle \sigma, T, \mathcal{P}[ \text{unprotected } e ] \rangle \mapsto_s \langle \sigma, T, \mathcal{P}[ \text{unprotected } e ], \text{unit} \rangle$$

for some  $\sigma$ ,  $T$ ,  $\mathcal{P}$ , and  $e$ . We say that this transition is an assertion violation if  $\mathcal{P}$  is of the form  $\mathcal{P}'[ \langle \mathcal{P}'' \rangle ]$ , for instance if  $\mathcal{P}[ \text{unprotected } e ]$  is

$$\langle \text{unprotected unit} \rangle$$

or

$$\langle \text{ref unprotected blockUntil true} \rangle$$

or

$$\text{ref } \langle \text{unprotected blockUntil true} \rangle$$

### 3.2.2 Typing States

We further generalize the type system to states  $\langle \sigma, T, e \rangle$ . We write

$$E; p_1 \cdots p_n, p \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$$

if

- $\text{dom}(\sigma) = \text{dom}(E) \cap \text{RefLoc}$ ,

- for all  $r \in \text{dom}(\sigma)$ , there exists  $t$  such that  $E(r) = \text{Ref } t$  and  $E; \text{NoYields} \vdash \sigma(r) : t$ ,
- $E; p_i \vdash e_i : \text{Unit}$  for all  $i = 1..n$ ,
- $E; p \vdash e : \text{Unit}$ .

We say that  $\langle \sigma, e_1. \dots .e_n, e \rangle$  is well-typed if there exist  $E$  and  $p_1, \dots, p_n, p$  such that  $E; p_1. \dots .p_n, p \vdash \langle \sigma, e_1. \dots .e_n, e \rangle$ .

### 3.2.3 Auxiliary Results

The first result is a replacement lemma, in the style of Wright and Felleisen. It immediately extends to typing states  $\langle \sigma, e_1. \dots .e_n, e \rangle$ .

**Lemma 3.1 (Replacement)** *Consider a derivation  $\mathcal{D}$  of  $E; p \vdash \mathcal{E}[e_0] : t$ . Assume that this derivation includes, as a subderivation, a proof  $\mathcal{D}_0$  of the judgment  $E; p_0 \vdash e_0 : t_0$  for the occurrence of  $e_0$  in  $\mathcal{E}[\cdot]$ . Assume that we also have a derivation  $\mathcal{D}'_0$  of  $E; p_0 \vdash e'_0 : t_0$  for some  $e'_0$ . Let  $\mathcal{D}'$  be obtained from  $\mathcal{D}$  by replacing  $\mathcal{D}_0$  with  $\mathcal{D}'_0$ , and  $e_0$  with  $e'_0$  in  $\mathcal{E}$ . Then  $\mathcal{D}'$  is a derivation of  $E; p \vdash \mathcal{E}[e'_0] : t$ .*

**Proof:** The proof is by induction on  $\mathcal{D}$ , as in the work of Wright and Felleisen. We omit the details. ■

The next results say that values can be typed as not yielding, if they can be typed at all, and that expressions that do not yield may be seen as yielding:

**Lemma 3.2** *If  $E; p \vdash V : t$  then  $E; \text{NoYields} \vdash V : t$ .*

**Proof:** The proof is by cases on the last rule being applied in the derivation of  $E; p \vdash V : t$ . In all the rules that can be used as the last one for typing a value ((Exp **unit**), (Exp **false**), (Exp **true**), (Exp  $x$ ), and (Exp **Fun**)), the type system leaves the choice of effect completely unconstrained. ■

**Lemma 3.3** *If  $E; \text{NoYields} \vdash e : t$  then  $E; \text{Yields} \vdash e : t$ .*

**Proof:** The proof is by induction on the derivation of  $E; \text{NoYields} \vdash e : t$ , with a case analysis on which rule is applied last. No rule forces a conclusion with **NoYields**: some rules where the conclusion may have effect **NoYields** (like (Exp **Async**) and (Exp **Assert**)) leave the choice of effect unconstrained,

while others (like (Exp Appl) and (Exp Ref)) propagate the effect used in the hypotheses of the rule application. In the latter case, `Yields` can be used instead of `NoYields` also in the hypotheses of the rule application, by induction hypothesis and, in the case of (Exp Appl), because  $q <: \text{Yields}$  always holds. ■

The next result is a standard substitution lemma.

**Lemma 3.4 (Substitution)** *If  $E, x : s, E' ; p \vdash e : t$  and  $E ; \text{NoYields} \vdash e' : s$  then  $E, E' ; p \vdash e[e'/x] : t$ .*

**Proof:** As usual, the proof is by induction on the derivation of  $E, x : s, E' ; p \vdash e : t$ . In the case of (Exp  $x$ ), we rely on Lemma 3.3. ■

Another lemma deals with updates to the state.

**Lemma 3.5** *Assume that  $r \in \text{dom}(\sigma)$  and  $E(r) = \text{Ref } t_0$ . If  $E ; p_1 \cdots p_n, p \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$  and  $E ; \text{NoYields} \vdash V : t_0$ , then  $E ; p_1 \cdots p_n, p \vdash \langle \sigma[r \mapsto V], e_1 \cdots e_n, e \rangle$ .*

**Proof:** This property follows directly from the definitions. ■

The final lemma provides an auxiliary fact on typing and unprotected expressions.

**Lemma 3.6** *It is never the case that  $E ; \text{NoYields} \vdash \mathcal{P}[\text{unprotected } e] : t$ .*

**Proof:** The proof is by induction on typing derivations, with a case analysis on which rule is applied last.

- The cases of (Exp Unit), (Exp Bool `false`), (Exp Bool `true`), (Exp  $x$ ), and (Exp Fun) are trivial, since the expressions typed there are values and cannot be the one in question.
- The case for (Exp Unprotect) is trivial because it gives an effect `Yields`.
- The cases of (Exp Appl), (Exp Ref), (Exp Deref), (Exp Set), and (Exp Block) are all by applications of the induction hypothesis, which are possible because the effects in the hypotheses of the rules are the same as the effects in their conclusions.
- The case for (Exp Async) is excluded because a context  $\mathcal{P}$  cannot be of the form `async  $\mathcal{P}'$` , so in this case  $\mathcal{P}$  must be `[]`, and `async  $\cdot$`  cannot match `unprotected  $e$` .

- The case for (Exp Assert) is by application of the induction hypothesis, since the effects in the hypothesis of the rule is **NoYields**.

■

### 3.2.4 Computation Preserves Typability

We obtain that typability is preserved by computation.

**Theorem 3.7 (Preservation of Typability)** *Suppose that  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ . If  $\langle \sigma, T, e \rangle$  is well-typed, then so is  $\langle \sigma', T', e' \rangle$ .*

**Proof:** We prove that if  $\langle \sigma, e_1 \cdots e_n, e \rangle$  is well-typed and  $\langle \sigma, e_1 \cdots e_n, e \rangle \mapsto_s \langle \sigma', e'_1 \cdots e'_{n'}, e' \rangle$  then  $\langle \sigma', e'_1 \cdots e'_{n'}, e' \rangle$  is well-typed. The theorem follows immediately by induction.

The proof is by cases on the operational-semantics rule being applied. In each case, we show that if

$$E; p_1 \cdots p_n, p \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$$

then

$$E'; p'_1 \cdots p'_{n'}, p' \vdash \langle \sigma', e'_1 \cdots e'_{n'}, e' \rangle$$

where, unless indicated otherwise,  $E' = E$ ,  $n' = n$ , and  $p'_i = p_i$  for  $i = 1..n$ . In several cases, we consider the typings of certain subexpressions that occur in evaluation contexts; those typings are with respect to  $E$ , since the holes in the contexts are never under binders.

- (Trans Appl)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[(\lambda x. e) V] \rangle$  must rely on (Exp Appl) and (Exp Fun). Specifically, we must have  $E; p_0 \vdash (\lambda x. e) V : t_0$  for some  $t_0$  and  $p_0$ , and therefore  $E; p_0 \vdash \lambda x. e : t_1 \rightarrow^{q_0} t_0$  for some  $q_0 <: p_0$  and  $E; p_0 \vdash V : t_1$  for some  $t_1$ , and therefore  $E, x : t_1; q_0 \vdash e : t_0$ . By Lemma 3.3,  $E, x : t_1; q_0 \vdash e : t_0$  and  $q_0 <: p_0$  imply  $E, x : t_1; p_0 \vdash e : t_0$ . By Lemma 3.4, we obtain  $E; p_0 \vdash e[V/x] : t_0$ . By Lemma 3.1, we obtain a typing of  $\langle \sigma, \mathcal{F}[e[V/x]] \rangle$ .
- (Trans Ref)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\mathbf{ref} V] \rangle$  must rely on (Exp Ref). Specifically, we must have  $E; p_0 \vdash \mathbf{ref} V : \mathbf{Ref} t_0$  for some  $t_0$  and  $p_0$ , and therefore  $E; p_0 \vdash V : t_0$ . By Lemma 3.2, we obtain  $E; \mathbf{NoYields} \vdash V : t_0$ . We extend  $E$  with  $r : \mathbf{Ref} t_0$ . We can do this extension because  $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$ , hence  $r \notin \mathit{dom}(E)$ . By a weakening (adding  $r : \mathbf{Ref} t_0$  to  $E$  for typing  $\langle \sigma, \mathcal{F}[\mathbf{ref} V] \rangle$ ) and Lemma 3.1, we obtain a typing of  $\langle \sigma, \mathcal{F}[r] \rangle$ .

- (Trans Deref)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[!r] \rangle$  must rely on (Exp Deref). Specifically, we must have  $E; p_0 \vdash !r : t_0$  for some  $t_0$  and  $p_0$ , and therefore  $E; p_0 \vdash r : \mathbf{Ref} \ t_0$ . Since  $r$  is a variable, its type must come from the environment  $E$ , so by hypothesis  $E; \mathbf{NoYields} \vdash V : t_0$  where  $V = \sigma(r)$ . By Lemma 3.3, we also have  $E; \mathbf{Yields} \vdash V : t_0$ , which is useful in case  $p_0$  is  $\mathbf{Yields}$ . By Lemma 3.1, we obtain a typing for  $\langle \sigma, \mathcal{F}[V] \rangle$ .
- (Trans Set)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[r := V] \rangle$  must rely on (Exp Set). Specifically, we must have  $E; p_0 \vdash r := V : \mathbf{Unit}$  for some  $p_0$ , and therefore  $E; p_0 \vdash V : t_0$  and  $E; p_0 \vdash r : \mathbf{Ref} \ t_0$  for some  $p_0$ . By Lemma 3.2,  $E; p_0 \vdash V : t_0$  implies  $E; \mathbf{NoYields} \vdash V : t_0$ . Since  $r$  is a variable, its type must come from the environment  $E$ . By Lemma 3.1, we can transform a typing of  $\langle \sigma, \mathcal{F}[r := V] \rangle$  into a typing of  $\langle \sigma, \mathcal{F}[\mathbf{unit}] \rangle$ , and since  $E; \mathbf{NoYields} \vdash V : t_0$  and  $E(r) = \mathbf{Ref} \ t_0$ , we also obtain a typing of  $\langle \sigma[r \mapsto V], \mathcal{F}[\mathbf{unit}] \rangle$  by Lemma 3.5.
- (Trans Async)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\mathbf{async} \ e] \rangle$  must rely on (Exp Async). Specifically, we must have  $E; p_0 \vdash \mathbf{async} \ e : \mathbf{Unit}$  for some  $p_0$ , and therefore that  $E; q_0 \vdash e : \mathbf{Unit}$  for some  $q_0$ . By Lemma 3.1, we can transform a typing of  $\langle \sigma, \mathcal{F}[\mathbf{async} \ e] \rangle$  into a typing of  $\mathcal{F}[\mathbf{unit}]$ , and then into a typing of  $\langle \sigma, e. \mathcal{F}[\mathbf{unit}] \rangle$  by letting  $n' = n + 1$  and adding  $q_0$  as first element to the sequence of effects.
- (Trans Block)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\mathbf{blockUntil} \ \mathbf{true}] \rangle$  must rely on (Exp Block), specifically on a derivation of  $E; p_0 \vdash \mathbf{blockUntil} \ \mathbf{true} : \mathbf{Unit}$  for some  $p_0$ . By Lemma 3.1, we obtain a typing of  $\langle \sigma, \mathcal{F}[\mathbf{unit}] \rangle$ .
- (Trans Unprotect)<sub>s</sub>: This case requires a trivial rearrangement in the effects:  $n' = n + 1$ ,  $p'_{n+1} = p$ , and  $p' = \mathbf{NoYields}$ .
- (Trans Close)<sub>s</sub>: The typing of  $\langle \sigma, T.\mathcal{E}[\mathbf{unprotected} \ V].T', e' \rangle$  must rely on (Exp Unprotect). Specifically, we must have  $E; \mathbf{Yields} \vdash \mathbf{unprotected} \ V : t_0$  for some  $t_0$ , and  $E; p_0 \vdash V : t_0$  for some  $p_0$ , so  $E; \mathbf{Yields} \vdash V : t_0$  by Lemma 3.3. By Lemma 3.1, we obtain a typing of  $\mathcal{E}[V]$  and then of  $\langle \sigma, T.\mathcal{E}[V].T', e' \rangle$ .
- (Trans Activate)<sub>s</sub>: This case requires a trivial rearrangement in the effects:  $n' = n - 1$ , and the effect  $p_i$  that corresponds to the expression  $e$  is skipped in  $p'_1 \cdots p'_{n'}$ , and becomes  $p'$ .
- (Trans Assert)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\langle V \rangle] \rangle$  must rely on (Exp Assert). Specifically, we must have  $E; p_0 \vdash \langle V \rangle : t_0$  for some  $t_0$  and  $p_0$ , and

$E; \text{NoYields} \vdash V : t_0$ , so  $E; p_0 \vdash V : t_0$  by Lemma 3.3. By Lemma 3.1, we obtain a typing of  $\mathcal{F}[V]$  and then of  $\langle \sigma, \mathcal{F}[V] \rangle$ .

■

The following corollary expresses the correctness of `NoYields`:

**Corollary 3.8** *If  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$  and  $\langle \sigma, T, e \rangle$  is well-typed, then none of the transitions in  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$  is an assertion violation.*

**Proof:** By Theorem 3.7, if  $\langle \sigma, T, e \rangle$  is well-typed then so are all the states reached in the computation  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ . Therefore, it suffices to prove that if  $\langle \sigma, T, e \rangle$  is well-typed and  $\langle \sigma, T, e \rangle \mapsto_s \langle \sigma', T', e' \rangle$ , then this transition is not an assertion violation. The claim in the theorem then follows by induction.

So suppose that  $\langle \sigma, T, e \rangle$  is well-typed and  $\langle \sigma, T, e \rangle \mapsto_s \langle \sigma', T', e' \rangle$ . The rule being applied could be  $(\text{Trans Unprotect})_s$ , with the rule application under an assertion, only if  $e$  is of the form  $\mathcal{P}'[ \langle \mathcal{P}''[ \text{unprotected } e' ] \rangle ]$  for some  $e'$ ,  $\mathcal{P}'$ , and  $\mathcal{P}''$ . If  $\langle \sigma, T, e \rangle$  is well-typed, then so is  $e$ , and therefore also  $\langle \mathcal{P}''[ \text{unprotected } e' ] \rangle$ , because a state can be well-typed only if all its components and their subexpressions are well-typed. By the typing rule for assertions, the fact that  $\langle \mathcal{P}''[ \text{unprotected } e' ] \rangle$  is well-typed implies that  $E'; \text{NoYields} \vdash \mathcal{P}''[ \text{unprotected } e' ] : t'$  for some  $E'$  and  $t'$ . We conclude by Lemma 3.6. ■

### 3.2.5 Progress

We also obtain a progress result, which characterizes when a computation may stop and implies that computations do not get stuck in unexpected ways (for instance, by applying a boolean as though it were a function).

**Lemma 3.9** *Suppose that  $e$  is a well-typed expression in which the only free variables are reference locations. Then  $e$  is a value or an expression of the form  $\mathcal{P}[f]$ , where  $f$  has one of the forms  $(\lambda x. e') V$ , `ref`  $V$ , `!r`, `r := V`, `async`  $e'$ , `blockUntil true`, `blockUntil false`, `unprotected`  $e'$ , and  $\langle V \rangle$ .*

**Proof:** The proof is by induction on the typing of  $e$ , with a case analysis on the last rule in the typing derivation.

- In the cases of `(Exp Unit)`, `(Exp Bool false)`, `(Exp Bool true)`, `(Exp x)`, and `(Exp Fun)`,  $e$  is a value.

- In the case of (Exp Appl),  $e$  cannot be a value. If  $e_1 e_2$  is well-typed, then  $e_1$  and  $e_2$  must be well-typed, and we apply the induction hypothesis to them. Suppose first that  $e_1$  is a value. Because the type of  $e_1$  must be a function type,  $e_1$  must be of the form  $\lambda x. e'$ . (It cannot be a variable because reference locations do not have function types.) If  $e_2$  is also a value  $V$ , we obtain that  $e$  is of the required form, with  $[\ ]$  for  $\mathcal{P}$ . If  $e_2$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $e_1 \mathcal{P}'$ . If  $e_1$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $\mathcal{P}' e_2$ .
- In the case of (Exp Ref),  $e$  cannot be a value. If  $\mathbf{ref} e_1$  is well-typed, then  $e_1$  must be well-typed, and we apply the induction hypothesis to it. Suppose first that  $e_1$  is a value. We obtain that  $e$  is of the required form, with  $[\ ]$  for  $\mathcal{P}$ . If  $e_1$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $\mathbf{ref} \mathcal{P}'$ .
- In the case of (Exp Deref),  $e$  cannot be a value. If  $!e_1$  is well-typed, then  $e_1$  must be well-typed, and we apply the induction hypothesis to it. Suppose first that  $e_1$  is a value. Because the type of  $e_1$  must be a reference type,  $e_1$  must be a reference location  $r$ . We obtain that  $e$  is of the required form, with  $[\ ]$  for  $\mathcal{P}$ . If  $e_1$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $!\mathcal{P}'$ .
- In the case of (Exp Set),  $e$  cannot be a value. If  $e_1 := e_2$  is well-typed, then  $e_1$  and  $e_2$  must be well-typed, and we apply the induction hypothesis to them. Suppose first that  $e_1$  is a value. Because the type of  $e_1$  must be a reference type,  $e_1$  must be a reference location  $r$ . If  $e_2$  is also a value  $V$ , we obtain that  $e$  is of the required form, with  $[\ ]$  for  $\mathcal{P}$ . If  $e_2$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $r := \mathcal{P}'$ . If  $e_1$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $\mathcal{P}' := e_2$ .
- The cases of (Exp Async) and (Exp Unprotect) are immediate, using the context  $[\ ]$ .
- In the case of (Exp Block),  $e$  cannot be a value. If  $\mathbf{blockUntil} e_1$  is well-typed, then  $e_1$  must be well-typed, and we apply the induction hypothesis to it. Suppose first that  $e_1$  is a value; according to the typing rules, it can be only **false** and **true**. (It cannot be a variable because reference locations do not have type `Bool`.) We obtain that  $e$  is of the required form, with  $[\ ]$  for  $\mathcal{P}$ . If  $e_1$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $\mathbf{blockUntil} \mathcal{P}'$ .

- In the case of (Exp Assert),  $e$  cannot be a value. If  $\langle e_1 \rangle$  is well-typed, then  $e_1$  must be well-typed, and we apply the induction hypothesis to it. Suppose first that  $e_1$  is a value. We obtain that  $e$  is of the required form, with  $[\ ]$  for  $\mathcal{P}$ . If  $e_1$  is not a value, then it is of the form  $\mathcal{P}'[f]$ , for an appropriate  $f$ , and we let  $\mathcal{P}$  be  $\langle \mathcal{P}' \rangle$ .

■

**Lemma 3.10** *If  $\langle \sigma, T, e \rangle$  is well-typed, and the only free variables in  $e$  are reference locations, then:*

1. *there exists  $\langle \sigma', T', e' \rangle$  such that  $\langle \sigma, T, e \rangle \mapsto_s \langle \sigma', T', e' \rangle$ ; or*
2.  *$e$  is of the form  $\mathcal{P}[\text{blockUntil false}]$ ; or*
3.  *$e$  is `unit` and  $T$  is empty.*

**Proof:** We apply Lemma 3.9 to  $e$ .

If  $e$  is a value, then it must be `unit` because  $\langle \sigma, T, e \rangle$  is well-typed and reference locations do not have type `Unit`. If  $T$  is empty, then we are in the third case. Otherwise, rule  $(\text{Trans Activate})_s$  applies, and we are in the first case.

If  $e$  is of the form  $\mathcal{P}[\text{blockUntil false}]$ , then we are immediately in the second case.

If  $e$  is of the form  $\mathcal{P}[f]$  where  $f$  is one of  $(\lambda x. e')$   $V$ , `ref`  $V$ ,  $!r$ ,  $r := V$ , `async`  $e'$ , `blockUntil true`, `unprotected`  $e'$ , or  $\langle V \rangle$  then, respectively,  $(\text{Trans Appl})_s$ ,  $(\text{Trans Ref})_s$ ,  $(\text{Trans Deref})_s$ ,  $(\text{Trans Set})_s$ ,  $(\text{Trans Async})_s$ ,  $(\text{Trans Block})_s$ ,  $(\text{Trans Unprotect})_s$ , or  $(\text{Trans Assert})_s$  apply, and we are in the first case again. In the case of  $(\text{Trans Ref})_s$ , we use that  $\text{RefLoc} - \text{dom}(\sigma)$  is never empty. In the case of  $(\text{Trans Deref})_s$ , we rely on the condition that if  $r \in \text{RefLoc}$  occurs free in  $e$  then  $r \in \text{dom}(\sigma)$ , and on the fact that  $\sigma$  maps reference locations to values. ■

**Theorem 3.11 (Progress)** *If  $\langle \sigma, T, e \rangle$  is well-typed, the only free variables in  $\langle \sigma, T, e \rangle$  are reference locations, and  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ , then:*

1. *there exists  $\langle \sigma'', T'', e'' \rangle$  such that  $\langle \sigma', T', e' \rangle \mapsto_s \langle \sigma'', T'', e'' \rangle$ ; or*
2.  *$e'$  is of the form  $\mathcal{P}[\text{blockUntil false}]$ ; or*
3.  *$e'$  is `unit` and  $T'$  is empty.*

---


$$\begin{array}{lcl}
s, t \in Type & = & \text{Unit} \\
& & | \text{Bool} \\
& & | s \rightarrow^p t \\
& & | \text{Ref}_p t \\
p, q \in \{P, U\} & & 
\end{array}$$

Figure 7: Types for separation.

---

**Proof:** According to Theorem 3.7, the state  $\langle \sigma', T', e' \rangle$  is well-typed. Since the rules of the operational semantics do not introduce free variables other than reference locations, the only free variables in  $e'$  are reference locations. The desired conclusion follows from Lemma 3.10. ■

This progress theorem is partly a sanity check. For instance, it does not exclude the possibility that a computation could get stuck going back and forth between two ill-typed states  $\langle \sigma, \text{unprotected false true}, \text{unit} \rangle$  and  $\langle \sigma, \emptyset, \text{unprotected false true} \rangle$ . Stronger progress theorems are viable.

## 4 A Type System for Separation

The type system described in this section embodies a discipline in which protected and unprotected computations do not use the same portions of the reference store. They may however communicate via variables. The type system is a rather simple static sufficient condition for some of our results, below.

Below, unless otherwise indicated, all references to typing and well-typing are with respect to the type system presented in this section.

### 4.1 The Type System

Figure 7 gives the syntax of types.

The type system is defined in Figure 8.

The type of an expression depends on a *typing environment*  $E$ , which maps variables to types. The typing environment is organized as a sequence of bindings, and we use  $\emptyset$  to denote the empty environment.

$$E ::= \emptyset \mid E, x : t$$

---

**Judgments**

$E \vdash \diamond$        $E$  is a well-formed typing environment  
 $E; p \vdash e : t$      $e$  is a well-typed expression of type  $t$  in  $E$  with  $p$

**Rules**

$\emptyset \vdash \diamond$	(Env $\emptyset$ )
$\frac{E \vdash \diamond \quad x \notin \text{dom}(E)}{E, x : t \vdash \diamond}$	(Env $x$ )
$\frac{E \vdash \diamond}{E; p \vdash \text{unit} : \text{Unit}}$	(Exp Unit)
$\frac{E \vdash \diamond}{E; p \vdash \text{false} : \text{Bool}}$	(Exp Bool <b>false</b> )
$\frac{E \vdash \diamond}{E; p \vdash \text{true} : \text{Bool}}$	(Exp Bool <b>true</b> )
$\frac{E, x : t, E' \vdash \diamond}{E, x : t, E'; p \vdash x : t}$	(Exp $x$ )
$\frac{E, x : s; p \vdash e : t}{E; q \vdash \lambda x. e : s \rightarrow^p t}$	(Exp Fun)
$\frac{E; p \vdash e_1 : s \rightarrow^p t \quad E; p \vdash e_2 : s}{E; p \vdash e_1 e_2 : t}$	(Exp Appl)
$\frac{E; p \vdash e : t}{E; p \vdash \text{ref } e : \text{Ref}_p t}$	(Exp Ref)
$\frac{E; p \vdash e : \text{Ref}_p t}{E; p \vdash !e : t}$	(Exp Deref)
$\frac{E; p \vdash e_1 : \text{Ref}_p t \quad E; p \vdash e_2 : t}{E; p \vdash e_1 := e_2 : \text{Unit}}$	(Exp Set)
$\frac{E; P \vdash e : \text{Unit}}{E; p \vdash \text{async } e : \text{Unit}}$	(Exp Async)
$\frac{E; p \vdash e : \text{Bool}}{E; p \vdash \text{blockUntil } e : \text{Unit}}$	(Exp Block)
$\frac{E; U \vdash e : t}{E; p \vdash \text{unprotected } e : t}$	(Exp Unprotect)

Figure 8: The first-order type system for separation.

---

We define the type system using judgments. These judgments are described in Figure 8, together with rules for reasoning about the judgments. The core of the type system is the set of rules for the judgment  $E; p \vdash e : t$  (read “ $e$  is a well-typed expression of type  $t$  in typing environment  $E$  with effect  $p$ ”). The intent is that, if this judgment holds, then  $e$  yields values of type  $t$  with effect  $p$ , and the free variables of  $e$  are given bindings consistent with the typing environment  $E$ . When  $p$  is  $P$ , this means that the evaluation of  $e$  accesses only the part of the reference store for protected computations; when  $p$  is  $U$ , this means that the evaluation of  $e$  accesses only the rest of the store. We say that  $e$  is well-typed when there exist  $E$ ,  $p$ , and  $t$  such that  $E; p \vdash e : t$ .

As explained above, the type system embodies a discipline in which protected and unprotected computations do not use the same portions of the reference store, but may communicate via variables. The following small example illustrates the restrictions that it imposes. Consider the small (and artificial) program:

```

let  $y = \text{ref true}$  in
  async ( $y := \text{false}$ ;
        unprotected  $z := !y$ );
  async (blockUntil  $!y$ ;
         $y := \text{true}$ )

```

This program is not permitted by the type system, because the reference location that is the value of  $y$  is used in both protected and unprotected computations. On the other hand, the following variant of the program is permitted by the type system:

```

let  $y = \text{ref true}$  in
  async ( $y := \text{false}$ ;
        let  $x = !y$  in (unprotected  $z := x$ ));
  async (blockUntil  $!y$ ;
         $y := \text{true}$ )

```

Here, the reference location in question is used only in protected computations; its value is put into a local variable  $x$  for use in an unprotected computation in the same thread.

## 4.2 Soundness of the Type System for Separation

This proof is adapted from the one for the type system for yielding. It is even easier, though. It should not be surprising. The proof has two

goals: (1) relate the type system to the absence of violations (defined in Section 5); (2) serve as a model for later proofs for a lower-level operational semantics. Although we do not always state them formally, some variants of these results hold with other lower-level semantics as well, in particular the weak semantics (for which the proofs are almost identical to those given here).

#### 4.2.1 Typing States

We extend the type system to states  $\langle \sigma, T, e \rangle$ . We write

$$E \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$$

if

- $dom(\sigma) = dom(E) \cap RefLoc$ ,
- for all  $r \in dom(\sigma)$ , there exist  $t$  and  $p$  such that  $E(r) = Ref_p t$  and  $E; p \vdash \sigma(r) : t$ ,
- $E; P \vdash e_i : \mathbf{Unit}$  for all  $i = 1..n$ ,
- $E; P \vdash e : \mathbf{Unit}$ .

We say that  $\langle \sigma, e_1 \cdots e_n, e \rangle$  is well-typed if there exist  $E$  such that  $E \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$ .

#### 4.2.2 Auxiliary Results

The first result is a replacement lemma, in the style of Wright and Felleisen. It immediately extends to typing states  $\langle \sigma, e_1 \cdots e_n, e \rangle$ .

**Lemma 4.1 (Replacement)** *Consider a derivation  $\mathcal{D}$  of  $E; p \vdash \mathcal{E}[e_0] : t$ . Assume that this derivation includes, as a subderivation, a proof  $\mathcal{D}_0$  of the judgment  $E; p_0 \vdash e_0 : t_0$  for the occurrence of  $e_0$  in  $\mathcal{E}[\cdot]$ . Assume that we also have a derivation  $\mathcal{D}'_0$  of  $E; p_0 \vdash e'_0 : t_0$  for some  $e'_0$ . Let  $\mathcal{D}'$  be obtained from  $\mathcal{D}$  by replacing  $\mathcal{D}_0$  with  $\mathcal{D}'_0$ , and  $e_0$  with  $e'_0$  in  $\mathcal{E}$ . Then  $\mathcal{D}'$  is a derivation of  $E; p \vdash \mathcal{E}[e'_0] : t$ .*

The next results say that values can be typed with either effect, if they can be typed at all.

**Lemma 4.2** *If  $E; p \vdash V : t$  then  $E; q \vdash V : t$ .*

The next result is a standard substitution lemma, restricted to values. (It may be overly general in allowing different  $p$  and  $q$ .)

**Lemma 4.3 (Value substitution)** *If  $E, x : s, E'; p \vdash e : t$  and  $E; q \vdash V : s$  then  $E, E'; p \vdash e[V/x] : t$ .*

**Proof:** As usual, the proof is by induction on the derivation of  $E, x : s, E'; p \vdash e : t$ . In the case of (Exp  $x$ ), we rely on Lemma 4.2. ■

Another lemma deals with updates to the state.

**Lemma 4.4** *Assume that  $r \in \text{dom}(\sigma)$  and  $E(r) = \text{Ref}_{p_0} t_0$ . If  $E \vdash \langle \sigma, e_1. \dots .e_n, e \rangle$  and  $E; p_0 \vdash V : t_0$ , then  $E \vdash \langle \sigma[r \mapsto V], e_1. \dots .e_n, e \rangle$ .*

The final lemma is a syntactic analysis of contexts.

**Lemma 4.5** *If  $E \vdash \langle \sigma, T, \mathcal{P}[e] \rangle$  then there exists  $t$  such that  $E; \mathbf{P} \vdash e : t$ . If  $E \vdash \langle \sigma, T, \mathcal{U}[e'] \rangle$  then there exists  $t$  such that  $E; \mathbf{U} \vdash e' : t$ .*

**Proof:** Since a state can be typed only if all of its components can be typed, if  $E \vdash \langle \sigma, T, \mathcal{P}[e] \rangle$  then there exist  $t$  and  $p$  such that  $E; p \vdash e : t$ , and if  $E \vdash \langle \sigma, T, \mathcal{U}[e'] \rangle$  then there exist  $t$  and  $p$  such that  $E; p \vdash e' : t$ . It remains to prove that  $p$  must be  $\mathbf{P}$  in the former case and  $\mathbf{U}$  in the latter case.

Generalizing from  $\mathbf{Unit}$  to an arbitrary  $t'$ , we show that if  $E; \mathbf{P} \vdash \mathcal{P}[e] : t'$  then  $E; \mathbf{P} \vdash e : t$  for some  $t$ , and that if  $E; \mathbf{P} \vdash \mathcal{U}[e'] : t'$  then  $E; \mathbf{U} \vdash e' : t$  for some  $t$ . We show in addition that if  $E; \mathbf{U} \vdash \mathcal{E}[e'] : t'$  then  $E; \mathbf{U} \vdash e' : t$  for some  $t$ .

The proofs are by induction on the forms of  $\mathcal{P}[e]$ ,  $\mathcal{U}[e']$ , and  $\mathcal{E}[e']$ .

- When  $\mathcal{P}[e]$  is  $e$ ,  $E; \mathbf{P} \vdash \mathcal{P}[e] : \mathbf{Unit}$  is  $E; \mathbf{P} \vdash e : \mathbf{Unit}$ . When  $\mathcal{P}[e]$  is of another form, the typing derivation of  $E; \mathbf{P} \vdash \mathcal{P}[e] : \mathbf{Unit}$  must rely on a proof of  $E; \mathbf{P} \vdash \mathcal{P}'[e] : \mathbf{Unit}$  for some smaller  $\mathcal{P}'[e]$ , according to the typing rules.
- Similarly, when  $\mathcal{E}[e']$  is  $e'$ , we are done immediately, and the cases for other forms are immediate applications of the induction hypothesis to smaller contexts.
- When  $\mathcal{U}[e']$  is unprotected  $\mathcal{E}[e']$ , we have  $E; \mathbf{U} \vdash \mathcal{E}[e'] : \mathbf{Unit}$ , and the previous case applies. For other forms of  $\mathcal{U}[e']$ , the cases are immediate applications of the induction hypothesis to smaller contexts.

■

### 4.2.3 Computation Preserves Typability

We obtain that typability is preserved by computation.

**Theorem 4.6 (Preservation of Typability)** *Suppose that  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ . If  $\langle \sigma, T, e \rangle$  is well-typed, then so is  $\langle \sigma', T', e' \rangle$ .*

**Proof:** We prove that if  $\langle \sigma, e_1 \dots e_n, e \rangle$  is well-typed and  $\langle \sigma, e_1 \dots e_n, e \rangle \mapsto_s \langle \sigma', e'_1 \dots e'_{n'}, e' \rangle$  then  $\langle \sigma', e'_1 \dots e'_{n'}, e' \rangle$  is well-typed. The first claim follows immediately by induction.

The proof is by cases on the operational-semantics rule being applied. In each case, we show that if  $E \vdash \langle \sigma, e_1 \dots e_n, e \rangle$  then  $E' \vdash \langle \sigma', e'_1 \dots e'_{n'}, e' \rangle$ , where, unless indicated otherwise,  $E' = E$  and  $n' = n$ . In several cases, we consider the typings of certain subexpressions that occur in evaluation contexts; those typings are with respect to  $E$ , since the holes in the contexts are never under binders.

- (Trans Appl)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[(\lambda x. e) V] \rangle$  must rely on (Exp Appl) and (Exp Fun). Specifically, we must have  $E; p_0 \vdash (\lambda x. e) V : t_0$  for some  $t_0$  and  $p_0$ , and therefore  $E; p_0 \vdash \lambda x. e : t_1 \xrightarrow{p_0} t_0$  and  $E; p_0 \vdash V : t_1$  for some  $t_1$ , and therefore  $E, x : t_1; p_0 \vdash e : t_0$ . By Lemma 4.3, we obtain  $E; p_0 \vdash e[V/x] : t_0$ . By Lemma 4.1, we obtain a typing of  $\langle \sigma, \mathcal{F}[e[V/x]] \rangle$ .
- (Trans Ref)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\mathbf{ref} V] \rangle$  must rely on (Exp Ref). Specifically, we must have  $E; p_0 \vdash \mathbf{ref} V : \mathbf{Ref}_{p_0} t_0$  for some  $t_0$  and  $p_0$ , and therefore  $E; p_0 \vdash V : t_0$ . We extend  $E$  with  $r : \mathbf{Ref}_{p_0} t_0$ . We can do this extension because  $r \in \mathit{RefLoc} - \mathit{dom}(\sigma)$ , hence  $r \notin \mathit{dom}(E)$ . By a weakening (adding  $r : \mathbf{Ref}_{p_0} t_0$  to  $E$  for typing  $\langle \sigma, \mathcal{F}[\mathbf{ref} V] \rangle$ ) and Lemma 4.1, we obtain a typing of  $\langle \sigma, \mathcal{F}[r] \rangle$ .
- (Trans Deref)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\mathbf{!}r] \rangle$  must rely on (Exp Deref). Specifically, we must have  $E; p_0 \vdash \mathbf{!}r : t_0$  for some  $t_0$  and  $p_0$ , and therefore  $E; p_0 \vdash r : \mathbf{Ref}_{p_0} t_0$ . Since  $r$  is a variable, its type must come from the environment  $E$ , so by hypothesis  $E; p_0 \vdash V : t_0$  where  $V = \sigma(r)$ . By Lemma 4.1, we obtain a typing for  $\langle \sigma, \mathcal{F}[V] \rangle$ .
- (Trans Set)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[r := V] \rangle$  must rely on (Exp Set). Specifically, we must have  $E; p_0 \vdash r := V : \mathbf{Unit}$  for some  $p_0$ , and therefore  $E; p_0 \vdash V : t_0$  and  $E; p_0 \vdash r : \mathbf{Ref}_{p_0} t_0$  for some  $p_0$ . By Lemma 4.1, we can transform a typing of  $\langle \sigma, \mathcal{F}[r := V] \rangle$  into a typing of  $\langle \sigma, \mathcal{F}[\mathbf{unit}] \rangle$ , and since  $E; p_0 \vdash V : t_0$  and  $E(r) = \mathbf{Ref}_{p_0} t_0$ , we also obtain a typing of  $\langle \sigma[r \mapsto V], \mathcal{F}[\mathbf{unit}] \rangle$  by Lemma 4.4.

- (Trans Async)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\text{async } e] \rangle$  must rely on (Exp Async). Specifically, we must have  $E; p_0 \vdash \text{async } e : \text{Unit}$  for some  $p_0$ , and therefore that  $E; \mathbf{P} \vdash e : \text{Unit}$ . By Lemma 4.1, we can transform a typing of  $\langle \sigma, \mathcal{F}[\text{async } e] \rangle$  into a typing of  $\mathcal{F}[\text{unit}]$ , and then into a typing of  $\langle \sigma, e.\mathcal{F}[\text{unit}] \rangle$ , letting  $n' = n + 1$ .
- (Trans Block)<sub>s</sub>: The typing of  $\langle \sigma, \mathcal{F}[\text{blockUntil true}] \rangle$  must rely on (Exp Block), specifically on a derivation of  $E; p_0 \vdash \text{blockUntil true} : \text{Unit}$  for some  $p_0$ . By Lemma 4.1, we obtain a typing of  $\langle \sigma, \mathcal{F}[\text{unit}] \rangle$ .
- (Trans Unprotect)<sub>s</sub>: This case requires a trivial rearrangement in the typing, with  $n' = n + 1$ .
- (Trans Close)<sub>s</sub>: The typing of  $\langle \sigma, T.\mathcal{E}[\text{unprotected } V].T', e' \rangle$  must rely on (Exp Unprotect). Specifically, we must have  $E; p_0 \vdash \text{unprotected } V : t_0$  for some  $t_0$  and  $p_0$ , and  $E; \mathbf{U} \vdash V : t_0$ , so  $E; p_0 \vdash V : t_0$  by Lemma 4.2. By Lemma 4.1, we obtain a typing of  $\mathcal{E}[V]$  and then of  $\langle \sigma, T.\mathcal{E}[V].T', e' \rangle$ .
- (Trans Activate)<sub>s</sub>: This case requires a trivial rearrangement in the typing, with  $n' = n - 1$ .

■

Examining the proof, we note that if  $\langle \sigma, T, e \rangle$  is well-typed with respect to an environment  $E$ , then  $\langle \sigma', T', e' \rangle$  is well-typed with respect to an extension of  $E$ . In the cases of (Trans Ref)<sub>s</sub>, (Trans Deref)<sub>s</sub>, and (Trans Set)<sub>s</sub>, which deal with a reference location of type  $\text{Ref}_{p_0} t_0$ , we also note that if the transition is protected, then  $p_0$  must be  $\mathbf{P}$ , and if the transition is unprotected, then  $p_0$  must be  $\mathbf{U}$ , by Lemma 4.5. It follows that, if  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$  and  $\langle \sigma, T, e \rangle$  is well-typed, then there exist subsets  $\mathbf{P}$  and  $\mathbf{U}$  of  $\text{dom}(\sigma')$  such that the protected transitions in  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$  allocate, read, or write only reference locations in  $\mathbf{P}$ , and the unprotected transitions in  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$  allocate, read, or write only reference locations in  $\mathbf{U}$ . The subsets in question consist of the reference locations declared with effects  $\mathbf{P}$  and  $\mathbf{U}$ , respectively, in the environment.

#### 4.2.4 Progress

We also obtain a progress result, which characterizes when a computation may stop and implies that computations do not get stuck in unexpected ways (for instance, by applying a boolean as though it were a function).

Like Theorem 3.11, this progress result is partly a sanity check; stronger ones are viable. It follows from the progress result for the type system for yielding and the following straightforward connection between the type systems:

**Lemma 4.7** *If  $\langle \sigma, T, e \rangle$  is well-typed in the type system for separation then it is also well-typed in the type system for yielding.*

**Proof:** Letting  $T = e_1 \cdots e_n$ , suppose that there exists  $E$  such that  $E \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$  in the type system for separation. Let  $E'$  be obtained from  $E$  by discarding information about P and U and using **Yields** as the only effect. By induction on typing derivations, a typing for separation can be turned into a typing for yielding with effect **Yields**. We obtain that

$$E' ; \mathbf{Yields} \cdots \mathbf{Yields}, \mathbf{Yields} \vdash \langle \sigma, e_1 \cdots e_n, e \rangle$$

(with  $n + 1$  occurrences of **Yields**) in the type system for yielding. The requirements on the reference store, which mention the effect **NoYields**, are satisfied by Lemma 3.2. ■

**Theorem 4.8 (Progress)** *If  $\langle \sigma, T, e \rangle$  is well-typed, the only free variables in  $\langle \sigma, T, e \rangle$  are reference locations, and  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ , then:*

1. *there exists  $\langle \sigma'', T'', e'' \rangle$  such that  $\langle \sigma', T', e' \rangle \mapsto_s \langle \sigma'', T'', e'' \rangle$ ; or*
2.  *$e'$  is of the form  $\mathcal{P}[\mathbf{blockUntil\ false} ]$ ; or*
3.  *$e'$  is **unit** and  $T'$  is empty.*

**Proof:** This theorem is an immediate consequence of Theorem 3.11 (the progress theorem for the type system for yielding) and Lemma 4.7 which relates the two type systems. ■

## 5 A Weaker Semantics

We show that, given a discipline for sharing, a weaker semantics (with more interleavings, and closer to an implementation) implements the strong semantics correctly.

We do this study without the rule (Trans Assert)<sub>s</sub> of Section 3.2.1, in order to keep the sections independent, but it would be trivial to include it.

For this section, we make a small technical restriction that does not affect the expressiveness of the language: in any expression of the form

---

**Evaluation contexts**

$$\mathcal{G} = T.\mathcal{U}.T', e' \mid T, \mathcal{P}$$

Figure 9: Additional evaluation contexts for weak semantics.

---

**Transition rules**

$\langle \sigma, \mathcal{G} [ (\lambda x. e) V ] \rangle$	$\mapsto_w \langle \sigma, \mathcal{G} [ e[V/x] ] \rangle$	(Trans Appl) <sub>w</sub>
$\langle \sigma, \mathcal{G} [ \text{ref } V ] \rangle$	$\mapsto_w \langle \sigma[r \mapsto V], \mathcal{G} [ r ] \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	(Trans Ref) <sub>w</sub>
$\langle \sigma, \mathcal{G} [ !r ] \rangle$	$\mapsto_w \langle \sigma, \mathcal{G} [ V ] \rangle$ if $\sigma(r) = V$	(Trans Deref) <sub>w</sub>
$\langle \sigma, \mathcal{G} [ r := V ] \rangle$	$\mapsto_w \langle \sigma[r \mapsto V], \mathcal{G} [ \text{unit} ] \rangle$	(Trans Set) <sub>w</sub>
$\langle \sigma, \mathcal{G} [ \text{async } e ] \rangle$	$\mapsto_w \langle \sigma, e. \mathcal{G} [ \text{unit} ] \rangle$	(Trans Async) <sub>w</sub>
$\langle \sigma, \mathcal{G} [ \text{blockUntil true} ] \rangle$	$\mapsto_w \langle \sigma, \mathcal{G} [ \text{unit} ] \rangle$	(Trans Block) <sub>w</sub>
$\langle \sigma, T, \mathcal{P} [ \text{unprotected } e ] \rangle$	$\mapsto_w \langle \sigma, T. \mathcal{P} [ \text{unprotected } e ], \text{unit} \rangle$	(Trans Unprotect) <sub>w</sub>
$\langle \sigma, T. \mathcal{E} [ \text{unprotected } V ]. T', e' \rangle$	$\mapsto_w \langle \sigma, T. \mathcal{E} [ V ]. T', e' \rangle$	(Trans Close) <sub>w</sub>
$\langle \sigma, T. e. T', \text{unit} \rangle$	$\mapsto_w \langle \sigma, T. T', e \rangle$	(Trans Activate) <sub>w</sub>

Figure 10: Transition rules of the abstract machine (weak).

---

`async e`, any occurrences of `unprotected` are under a  $\lambda$ . Thus, using our abbreviations, we can write `async (unit; unprotected e')`, but not `async (unprotected e')`. More generally, we can write `async (unit; e')`, for any  $e'$ . This technical restriction roughly ensures that an unprotected computation is not the first thing that happens in an asynchronous computation.

## 5.1 The Weak Semantics

Figures 9 and 10 define the weak semantics. The subscript  $w$  in  $\mapsto_w$  indicates that this is a weak semantics, in the sense that it permits interleavings

of executions of atomic fragments and unprotected fragments, as discussed further below.

As indicated in Figure 9,  $\mathcal{G}$  evaluation contexts are general evaluation contexts, defined so that  $\mathcal{G}[e]$  includes both  $T\mathcal{U}[e].T', e'$  and  $T, \mathcal{P}[e]$ . Since  $\mathcal{G}[e]$  is either  $T\mathcal{U}[e].T', e'$  or  $T, \mathcal{P}[e]$ , we write  $e_0.\mathcal{G}[e_1]$  as an abbreviation for  $e_0.T\mathcal{U}[e_1].T', e'$  or  $e_0.T, \mathcal{P}[e_1]$ , respectively.

Consider a transition  $\langle \sigma, e_1 \cdots e_n, e \rangle \mapsto_w \langle \sigma', e'_1 \cdots e'_{n'}, e' \rangle$ . Unless it is an instance of  $(\text{Trans Unprotect})_w$  or  $(\text{Trans Activate})_w$ , the transition is defined in terms of a context that has a hole either in  $e_1 \cdots e_n$  and in  $e'_1 \cdots e'_{n'}$ , or in  $e$  and in  $e'$ . We say that the transition is protected if the hole is in  $e$  and in  $e'$ , and say that the transition is unprotected if the hole is in  $e_1 \cdots e_n$  and in  $e'_1 \cdots e'_{n'}$ . By definition, transitions that are instances of  $(\text{Trans Close})_w$  are always unprotected; transitions that are instances of  $(\text{Trans Unprotect})_w$  or  $(\text{Trans Activate})_w$  are neither protected nor unprotected.

We have  $\langle \sigma, T, e \rangle \mapsto_s \langle \sigma', T', e' \rangle$  when  $\langle \sigma, T, e \rangle \mapsto_w \langle \sigma', T', e' \rangle$  and, if this transition is unprotected, then  $e = \text{unit}$ .

## 5.2 A Hypothesis

We formalize a hypothesis according to which data cannot be accessed with and without protection at the same time in different threads.

Given a state  $\langle \sigma, e_1 \cdots e_n, e \rangle$ , there is a violation on a location  $r$  if  $e_i = \mathcal{U}[f]$  for some  $i = 1..n$  and  $e = \mathcal{P}[f']$  where  $f$  and  $f'$  are reads or writes on  $r$  (that is, expressions  $!r$  or  $r := \dots$ ), and at least one of them is a write ( $r := \dots$ ). A computation is violation-free if none of its states have violations for any locations. (Analogously, we could define races, in which we would also consider conflicts within  $e_1 \cdots e_n$ ; every violation is a race but not every race is a violation.)

A possible programming discipline consists in requiring that programs never generate violations in the strong semantics. According to this discipline, a state  $\langle \sigma, T, e \rangle$  is good if all strong computations that start from this state are violation-free.

The use of the strong semantics is significant: programmers should not have to understand lower-level implementations. However, analogous criteria apply to lower-level implementations, and might be of benefit in compiler optimizations. Sometimes (but not always!) we obtain results that say that the absence of violations in the strong semantics automatically implies the absence of violations in lower-level implementations; see for example Lemma 6.5.

This discipline relies on a simple but undecidable dynamic criterion. While the type system for separation is (deliberately) simplistic, it provides a sufficient condition for enforcing the discipline. The type system constitutes a stronger discipline, and appears to be more robust (for instance, less fragile in the presence of compiler optimizations).

**Corollary 5.1** *If  $\langle \sigma, T, e \rangle$  is well-typed, then all strong computations that start from  $\langle \sigma, T, e \rangle$  are violation-free.*

**Proof:** Suppose that  $\langle \sigma, T, e \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ . By Theorem 4.6, if  $\langle \sigma, T, e \rangle$  is well-typed then so is  $\langle \sigma', T', e' \rangle$ , with some typing environment  $E'$ . If there is a violation on a location  $r$  in  $\langle \sigma', T', e' \rangle$ , then  $T'$  includes an expression of the form  $\mathcal{U}[f]$  and  $e'$  is of the form  $\mathcal{P}[f']$ , where  $f$  and  $f'$  are reads or writes on  $r$ . By Lemma 4.5, there exists  $t$  such that  $E'; \mathbb{U} \vdash f : t$  and  $t'$  such that  $E'; \mathbb{P} \vdash f' : t'$ . So, according to the typing rules,  $r$  must have a type of the form  $\text{Ref}_{\mathbb{U}} t_0$  in  $E'$  and also a type of the form  $\text{Ref}_{\mathbb{P}} t_0$  in  $E'$ . This conclusion contradicts the well-formedness of  $E'$ . ■

### 5.3 Correctness

We prove a commutation lemma:

**Lemma 5.2** *Suppose that  $\langle \sigma, T, e \rangle \mapsto_w \langle \sigma', T', e' \rangle \mapsto_w \langle \sigma'', T'', e'' \rangle$  where the first transition is an instance of  $(\text{Trans Activate})_w$  or a protected transition and the second transition is an unprotected transition. Suppose further that  $\langle \sigma, T, e \rangle$  is violation-free. Then there exists  $\langle \sigma^*, T^*, e^* \rangle$  such that  $\langle \sigma, T, e \rangle \mapsto_w \langle \sigma^*, T^*, e^* \rangle \mapsto_w \langle \sigma'', T'', e'' \rangle$  where the second transition is an instance of  $(\text{Trans Activate})_w$  or a protected transition.*

**Proof:** Consider a transition that is an instance of  $(\text{Trans Activate})_w$  followed by an unprotected transition:

$$\langle \sigma_k, e_1 \cdots e_i \cdots e_n.e, \text{unit} \rangle \mapsto_w \langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}, e_1 \cdots e'_i \cdots e_n, e \rangle$$

(We place  $e$  at the end of  $e_1 \cdots e_i \cdots e_n.e$  only as a notational convenience; the argument is analogous in other cases.) We can rewrite this pair of transitions so that the unprotected transition is before the instance of  $(\text{Trans Activate})_w$ :

$$\langle \sigma_k, e_1 \cdots e_i \cdots e_n.e, \text{unit} \rangle \mapsto_w \langle \sigma_{k+1}, e_1 \cdots e'_i \cdots e_n.e, \text{unit} \rangle \mapsto_w \langle \sigma_{k+1}, e_1 \cdots e'_i \cdots e_n, e \rangle$$

The unprotected transition uses the same rule as before the commutation, with only a straightforward change in the choice of evaluation context that corresponds to the change from  $e_1 \cdots e_i \cdots e_n, e$  to  $e_1 \cdots e_i \cdots e_n, e, \mathbf{unit}$ .

Consider a protected transition followed by an unprotected transition:

$$\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}, e_1 \cdots e_i \cdots e_n, e' \rangle \mapsto_w \langle \sigma_{k+2}, e_1 \cdots e'_i \cdots e_n, e' \rangle$$

where the protected transition is not an instance of  $(\text{Trans Async})_w$ . We can rewrite this pair of transitions so that the unprotected transition is before the protected transition:

$$\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}^*, e_1 \cdots e'_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+2}, e_1 \cdots e'_i \cdots e_n, e' \rangle$$

for an appropriate  $\sigma_{k+1}^*$ , which is constructed by case analysis on the protected transition:

- In the cases of  $(\text{Trans Appl})_w$  and  $(\text{Trans Block})_w$ , we have that  $\sigma_{k+1} = \sigma_k$ , and we let  $\sigma_{k+1}^* = \sigma_{k+2}$ .
- In the case of  $(\text{Trans Ref})_w$ , we have that  $\sigma_{k+1} = \sigma_k[r \mapsto V]$  for some  $r$  and  $V$ . This location  $r$  is required not to be in  $\text{dom}(\sigma_k)$ , and hence (by our general requirements on states) cannot occur free in  $e_i$ . Therefore, the unprotected transition cannot read or write the contents of  $r$ , so  $\sigma_{k+2}(r) = V$ . Moreover, if the unprotected transition is itself an instance of  $(\text{Trans Ref})_w$ , it cannot allocate the same  $r$ . We let  $\sigma_{k+1}^*$  be the restriction of  $\sigma_{k+2}$  to locations other than  $r$ . We have  $\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}^*, e_1 \cdots e'_i \cdots e_n, e \rangle$  because the unprotected transition does not allocate, read, or write  $r$ . We have  $\langle \sigma_{k+1}^*, e_1 \cdots e'_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+2}, e_1 \cdots e'_i \cdots e_n, e' \rangle$  because the unprotected transition does not allocate or write  $r$ .
- In the case of  $(\text{Trans Deref})_w$ , we have that  $\sigma_{k+1} = \sigma_k$ , and we let  $\sigma_{k+1}^* = \sigma_{k+2}$ . Thus,  $\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}^*, e_1 \cdots e'_i \cdots e_n, e \rangle$ . Because  $\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle$  is violation-free, this unprotected transition cannot be a write to the location  $r$  read in  $(\text{Trans Deref})_w$ , so  $\sigma_{k+2}(r) = \sigma_k(r)$  and hence  $\langle \sigma_{k+1}^*, e_1 \cdots e'_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+2}, e_1 \cdots e'_i \cdots e_n, e' \rangle$ .
- In the case of  $(\text{Trans Set})_w$ , we have that  $\sigma_{k+1} = \sigma_k[r \mapsto V]$  for some  $r$  and  $V$ . By our general requirements on states, we have that  $r \in \text{dom}(\sigma_k)$ . We let  $\sigma_{k+1}^* = \sigma_{k+2}[r \mapsto V_0]$  where  $V_0 = \sigma_k(r)$ . Because  $r \in \text{dom}(\sigma_{k+1})$ , the unprotected transition cannot allocate  $r$ , and because  $\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle$  is violation-free, the unprotected transition cannot read or write  $r$ , so  $\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}^*, e_1 \cdots e'_i \cdots e_n, e \rangle$ .

In addition, we obtain  $\langle \sigma_{k+1}^*, e_1 \cdots e_i' \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+2}, e_1 \cdots e_i' \cdots e_n, e' \rangle$  from  $\sigma_{k+2} = \sigma_{k+1}^*[r \mapsto V]$  and by the definition of the transition relation in the case of  $(\text{Trans Set})_w$ .

Similarly, consider a protected transition followed by an unprotected transition:

$$\langle \sigma_k, e_1 \cdots e_i \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}, e'' \cdot e_1 \cdots e_i \cdots e_n, e' \rangle \mapsto_w \langle \sigma_{k+2}, e'' \cdot e_1 \cdots e_i' \cdots e_n, e' \rangle$$

where the protected transition is an instance of  $(\text{Trans Async})_w$ . As in the cases of  $(\text{Trans Appl})_w$  and  $(\text{Trans Block})_w$ , we have that  $\sigma_{k+1} = \sigma_k$ , and we let  $\sigma_{k+1}^* = \sigma_{k+2}$ . Note that we cannot have:

$$\langle \sigma_k, e_1 \cdots e_n, e \rangle \mapsto_w \langle \sigma_{k+1}, e'' \cdot e_1 \cdots e_n, e' \rangle \mapsto_w \langle \sigma_{k+2}, e''' \cdot e_1 \cdots e_n, e' \rangle$$

where the first transition is an instance of  $(\text{Trans Async})_w$  and the second transition operates immediately on the expression  $e''$  added to the pool: this possibility is excluded by the restriction that in, an expression of the form `async`  $e''$ , any occurrences of `unprotected` are under a  $\lambda$ . ■

We obtain the following correctness theorem.

**Theorem 5.3** *Assume that all strong computations that start from the state  $\langle \sigma, T, \text{unit} \rangle$  are violation-free. Consider a weak computation  $\langle \sigma, T, \text{unit} \rangle \mapsto_w^* \langle \sigma', T', e' \rangle$ . Then there is a strong computation  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ .*

**Proof:** The proof is by induction on the length of  $\langle \sigma, T, \text{unit} \rangle \mapsto_w^* \langle \sigma', T', e' \rangle$ . We strengthen the claim with the assertion that the strong computation is of the same length as the original computation.

The claim is vacuously true when this length is 0. For the inductive step, suppose that we have  $\langle \sigma, T, \text{unit} \rangle \mapsto_w^* \langle \sigma', T', e' \rangle \mapsto_w \langle \sigma'', T'', e'' \rangle$ . By induction hypothesis,  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma', T', e' \rangle \mapsto_w \langle \sigma'', T'', e'' \rangle$ . If  $\langle \sigma', T', e' \rangle \mapsto_s \langle \sigma'', T'', e'' \rangle$ , then we immediately obtain a strong computation  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma'', T'', e'' \rangle$ . On the other hand, if  $\langle \sigma', T', e' \rangle \mapsto_w \langle \sigma'', T'', e'' \rangle$  is not strong, then there must be a transition before it (since otherwise  $e' = e = \text{unit}$ ), in other words a last transition in  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ . Moreover this strong transition to  $\langle \sigma', T', e' \rangle$  cannot be an instance of  $(\text{Trans Unprotect})_w$  nor an unprotected strong transition, for otherwise  $e' = \text{unit}$ , so it must be an instance of  $(\text{Trans Activate})_w$  or a protected transition. By Lemma 5.2, we can commute steps in order to obtain a strong computation  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma'', T'', e'' \rangle$ . Specifically, Lemma 5.2 shows how to move an unprotected transition before an instance of  $(\text{Trans$

Activate) $_w$  or a protected transition. By induction hypothesis, the computation up to this unprotected transition can be made strong, and adding back the instance of (Trans Activate) $_w$  or the protected transition yields the desired strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma'', T'', e'' \rangle$ .

Note that the new intermediate states generated in the proof could have violations, and the argument would collapse, if we had not assumed that all strong computations that start from the state  $\langle \sigma, T, \mathbf{unit} \rangle$  are violation-free. Considering strong computations is fortunately enough; considering only the given computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_w^* \langle \sigma', T', e' \rangle$  would not be. ■

We deduce a correctness theorem with a static check as hypothesis:

**Corollary 5.4** *Assume that  $\langle \sigma, T, \mathbf{unit} \rangle$  is well-typed. Consider a weak computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_w^* \langle \sigma', T', e' \rangle$ . Then there is a strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma', T', e' \rangle$ .*

**Proof:** This is an immediate consequence of Corollary 5.1 and Theorem 5.3. ■

## 6 Roll-back

In our high-level semantics, we do not model the workings of roll-backs. A straightforward implementation of our language that includes roll-backs may not respect the semantics for all programs. Indeed, in weak semantics where unprotected transitions can be interleaved with protected transitions arbitrarily, the unprotected transitions may be able to see intermediate states of a roll-back; upon a roll-back, there is the question of whether unprotected computations should be rolled-back as well. For programs that do not cause violations, however, Theorem 5.3 suggests that discarding the effects of protected transitions can be a correct implementation, as we explain in this section. We develop lower-level semantics with roll-back and establish their correctness.

We focus on the roll-backs that one may wish to perform when a computation is blocked on a false condition (that is, on roll-backs caused when a thread decides to abort and retry later). Other kinds of roll-backs (in particular, those used in the implementation of optimistic concurrency schemes) may be treated similarly, with the possible addition of machinery for conflict detection. We consider those in Section 7.

## 6.1 An Informal Account

The results on strong computations enable us to give a simple account of roll-backs.

Suppose that we have a computation

$$\langle \sigma, T, \mathbf{unit} \rangle \mapsto_w^* \langle \sigma', T', \mathcal{P}[\mathbf{blockUntil\ false}] \rangle$$

where only the first transition is an instance of  $(\text{Trans Activate})_w$ . We may wish to roll back the computation, restoring the state to  $\langle \sigma, T, \mathbf{unit} \rangle$  entirely or partly. The details of how this is done vary across implementations. They can be problematic if the computation includes unprotected transitions, possibly with corresponding modifications to the store. The results of roll-backs are sometimes surprising.

Because of Theorem 5.3, however, we have a strong computation

$$\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma', T', \mathcal{P}[\mathbf{blockUntil\ false}] \rangle$$

This strong computation also has exactly one instance of  $(\text{Trans Activate})_w$ , but it will not in general be the first transition. All unprotected transitions must be before this instance. Therefore, rolling back exactly to the state before the application of  $(\text{Trans Activate})_w$  is a clean approach.

## 6.2 A Lower-Level Semantics with Roll-back

We model an optimistic scheme based on undos. Other flavors of roll-back are possible. One goal of the semantics without roll-back is to abstract from the details of particular implementations of roll-back, such as those we show here; it should not be surprising that those details can be somewhat cumbersome and arbitrary.

### 6.2.1 States

As described in Figure 11, a state  $\langle \sigma, T, e, f, l, P \rangle$  consists of the following components:

- $\sigma$ ,  $T$ , and  $e$ , which are as usual,
- $f$ , an expression that, through computation, has yielded  $e$  (which we call the origin of  $e$ ),
- $l$ , a list of memory locations and their values, to be used as a log in undos,

---

**State space**

$$\begin{aligned} S &\in \text{State} = \text{RefStore} \times \text{ExpSeq} \times \text{Exp} \times \text{Exp} \times \text{Log} \times \text{ExpSeq} \\ \sigma &\in \text{RefStore} = \text{RefLoc} \rightarrow \text{Value} \\ l &\in \text{Log} = (\text{RefLoc} \times \text{Value})^* \\ r &\in \text{RefLoc} \subset \text{Var} \\ T, P &\in \text{ExpSeq} = \text{Exp}^* \end{aligned}$$

Figure 11: State space, with roll-back.

---

- $P$ , a list of threads to be forked upon commit.

We write each pair in  $l$  in the form  $[r \mapsto V]$ , we let  $\text{dom}(l)$  be the set of locations  $r$  for which  $l$  is defined, and when  $r \in \text{dom}(l)$  we write  $l(r)$  for the value  $V$  to which  $r$  is mapped.

Much as in Section 2.1, for every state  $\langle \sigma, T, e, f, l, P \rangle$ , we require that if  $r \in \text{RefLoc}$  occurs free in  $\sigma(r')$ , in  $T$ , in  $e$ , in  $f$ , in  $l$ , or in  $P$ , then  $r \in \text{dom}(\sigma)$ . This condition will be assumed for initial states and will be preserved by computation steps.

### 6.2.2 Steps: Weak Semantics with Roll-back

Figure 12 gives the rules of a weak semantics with roll-back. We do not use contexts  $\mathcal{G}$  because they are not useful in most of the rules.

The intent is that, upon a roll-back caused by  $e$ , the origin expression  $f$  is added back to  $T$  and the undos described in  $l$  are performed. Note that:

- the undos described in  $l$  are not performed in one atomic step;
- allocations are not undone; if they were, we could cause dangling pointers in programs with race conditions;
- after roll-back,  $e$  is in general not the active expression, though it may become the active expression some time later;
- no undo facilities are provided for unprotected computations.

In this semantics, each transition has the form

$$\langle \sigma, T, e, f, l, P \rangle \mapsto_{rw} \langle \sigma', T', e', f', l', P' \rangle$$

In many cases, a transition is defined in terms of a context that has a hole either in  $T$  and in  $T'$ , or in  $e$  and in  $e'$ . We say that the transition is protected

---

**Transition rules**

$\langle \sigma, T, \mathcal{P} [ (\lambda x. e) V ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{P} [ e[V/x] ], f, l, P \rangle$	(Trans Appl P) <sub>rw</sub>
$\langle \sigma, T, \mathcal{U} [ (\lambda x. e) V ].T', e', f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{U} [ e[V/x] ].T', e', f, l, P \rangle$	(Trans Appl U) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ \text{ref } V ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{P} [ r ], f, l, P \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	(Trans Ref P) <sub>rw</sub>
$\langle \sigma, T, \mathcal{U} [ \text{ref } V ].T', e, f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{U} [ r ].T', e, f, l, P \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	(Trans Ref U) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ !r ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{P} [ V ], f, l, P \rangle$ if $\sigma(r) = V$	(Trans Deref P) <sub>rw</sub>
$\langle \sigma, T, \mathcal{U} [ !r ].T', e, f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{U} [ V ].T', e, f, l, P \rangle$ if $\sigma(r) = V$	(Trans Deref U) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ r := V ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{P} [ \text{unit} ], f, l', P \rangle$ where $l' = \text{if } r \in \text{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)]$	(Trans Set P) <sub>rw</sub>
$\langle \sigma, T, \mathcal{U} [ r := V ].T', e, f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{U} [ \text{unit} ].T', e, f, l, P \rangle$	(Trans Set U) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ \text{async } e ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{P} [ \text{unit} ], f, l, e.P \rangle$	(Trans Async P) <sub>rw</sub>
$\langle \sigma, T, \mathcal{U} [ \text{async } e ].T', e', f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, e.T, \mathcal{U} [ \text{unit} ].T', e', f, l, P \rangle$	(Trans Async U) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ \text{blockUntil true} ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{P} [ \text{unit} ], f, l, P \rangle$	(Trans Block true P) <sub>rw</sub>
$\langle \sigma, T, \mathcal{U} [ \text{blockUntil true} ].T', e, f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{U} [ \text{unit} ].T', e, f, l, P \rangle$	(Trans Block true U) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ \text{blockUntil false} ], f, \emptyset, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, f.T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	(Trans Block false Restore) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ \text{blockUntil false} ], f, l.[r \mapsto V], P \rangle$	$\mapsto_{rw}$	$\langle \sigma[r \mapsto V], T, \mathcal{P} [ \text{blockUntil false} ], f, l, P \rangle$	(Trans Block false Undo) <sub>rw</sub>
$\langle \sigma, T, \mathcal{P} [ \text{unprotected } e ], f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T.P, \mathcal{P} [ \text{unprotected } e ], \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	(Trans Unprotect) <sub>rw</sub>
$\langle \sigma, T, \text{unit}, f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T.P, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	(Trans Done) <sub>rw</sub>
$\langle \sigma, T, \mathcal{E} [ \text{unprotected } V ].T', e, f, l, P \rangle$	$\mapsto_{rw}$	$\langle \sigma, T, \mathcal{E} [ V ].T', e, f, l, P \rangle$	(Trans Close) <sub>rw</sub>
$\langle \sigma, T.e.T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rw}$	$\langle \sigma, T.T', e, e, \emptyset, \emptyset \rangle$	(Trans Activate) <sub>rw</sub>

Figure 12: Transition rules of the abstract machine, with roll-back (weak).

---

---

**Transition rules**

$\langle \sigma, T, \mathcal{P} [ (\lambda x. e) V ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{P} [ e[V/x] ], f, l, P \rangle$	$(\text{Trans Appl P})_{rs}$
$\langle \sigma, T, \mathcal{M} [ (\lambda x. e) V ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{M} [ e[V/x] ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Appl U})_{rs}$
$\langle \sigma, T, \mathcal{P} [ \text{ref } V ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma[r \mapsto V], T, \mathcal{P} [ r ], f, l, P \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	$(\text{Trans Ref P})_{rs}$
$\langle \sigma, T, \mathcal{M} [ \text{ref } V ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma[r \mapsto V], T, \mathcal{M} [ r ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	$(\text{Trans Ref U})_{rs}$
$\langle \sigma, T, \mathcal{P} [ !r ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{P} [ V ], f, l, P \rangle$ if $\sigma(r) = V$	$(\text{Trans Deref P})_{rs}$
$\langle \sigma, T, \mathcal{M} [ !r ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{M} [ V ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$ if $\sigma(r) = V$	$(\text{Trans Deref U})_{rs}$
$\langle \sigma, T, \mathcal{P} [ r := V ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma[r \mapsto V], T, \mathcal{P} [ \text{unit} ], f, l', P \rangle$ where $l' = \text{if } r \in \text{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)]$	$(\text{Trans Set P})_{rs}$
$\langle \sigma, T, \mathcal{M} [ r := V ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma[r \mapsto V], T, \mathcal{M} [ \text{unit} ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Set U})_{rs}$
$\langle \sigma, T, \mathcal{P} [ \text{async } e ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{P} [ \text{unit} ], f, l, e.P \rangle$	$(\text{Trans Async P})_{rs}$
$\langle \sigma, T, \mathcal{M} [ \text{async } e ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma, e.T, \mathcal{M} [ \text{unit} ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Async U})_{rs}$
$\langle \sigma, T, \mathcal{P} [ \text{blockUntil true} ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{P} [ \text{unit} ], f, l, P \rangle$	$(\text{Trans Block true P})_{rs}$
$\langle \sigma, T, \mathcal{M} [ \text{blockUntil true} ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{M} [ \text{unit} ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Block true U})_{rs}$
$\langle \sigma, T, \mathcal{P} [ \text{blockUntil false} ], f, \emptyset, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, f.T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Block false Restore})_{rs}$
$\langle \sigma, T, \mathcal{P} [ \text{blockUntil false} ], f, l.[r \mapsto V], P \rangle$	$\mapsto_{rs}$	$\langle \sigma[r \mapsto V], T, \mathcal{P} [ \text{blockUntil false} ], f, l, P \rangle$	$(\text{Trans Block false Undo})_{rs}$
$\langle \sigma, T, \mathcal{P} [ \text{unprotected } e ], f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, T.P, \mathcal{P} [ \text{unprotected } e ], \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Unprotect})_{rs}$
$\langle \sigma, T, \mathcal{E} [ \text{unprotected } V ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma, T, \mathcal{E} [ V ].T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Close})_{rs}$
$\langle \sigma, T, \text{unit}, f, l, P \rangle$	$\mapsto_{rs}$	$\langle \sigma, T.P, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$(\text{Trans Done})_{rs}$
$\langle \sigma, T.e.T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$	$\mapsto_{rs}$	$\langle \sigma, T.T', e, e, \emptyset, \emptyset \rangle$	$(\text{Trans Activate})_{rs}$

Figure 13: Transition rules of the abstract machine, with roll-back (strong).

---

if the hole is in  $e$  and in  $e'$ , and say that the transition is unprotected if the hole is in  $T$  and in  $T'$ . By definition, we have:

- transitions that are instances of  $(\text{Trans } \dots \text{P})_{rw}$  or  $(\text{Trans Block false Undo})_{rw}$  are always protected;
- transitions that are instances of  $(\text{Trans } \dots \text{U})_{rw}$  or of  $(\text{Trans Close})_{rw}$  are always unprotected;
- transitions that are instances of  $(\text{Trans Unprotect})_{rw}$ ,  $(\text{Trans Block false Restore})_{rw}$ ,  $(\text{Trans Done})_{rw}$ , or  $(\text{Trans Activate})_{rw}$  are neither protected nor unprotected.

### 6.2.3 Steps: Strong Semantics with Roll-back

Figure 13 gives the rules of a strong semantics with roll-back. (We do not use contexts  $\mathcal{F}$  because they are not useful in most of the rules, much as above.)

We have

$$\langle \sigma, T, e, f, l, P \rangle \mapsto_{rs} \langle \sigma', T', e', f', l', P' \rangle$$

when  $\langle \sigma, T, e, f, l, P \rangle \mapsto_{rw} \langle \sigma', T', e', f', l', P' \rangle$  and, if this transition is unprotected, then  $e = f = \mathbf{unit}$ ,  $l = \emptyset$ , and  $P = \emptyset$ . We say that this transition is a strong transition.

## 6.3 Correctness

The goal of this section is to show that semantics with roll-back (which are closer to actual realizations of the language) are correct implementations of the simpler, higher-level semantics without roll-back. More specifically, we show that the weak semantics with roll-back implements the strong semantics without roll-back, assuming the absence of violations.

In addition, we prove that the strong semantics with roll-back implements the strong semantics without roll-back. This result does not require the absence of violations. On the other hand, the weak semantics with roll-back is not a correct implementation of the weak semantics without roll-back unless one makes some assumptions. Intuitively, in weak semantics, violations can allow unprotected computations to observe intermediate states of computations to be undone. It appears, then, that the weak semantics without roll-back is not so attractive: it is hard to implement in the presence of violations, and, when violations are disallowed, it is equivalent to the simpler strong semantics. Accordingly, in our intermediate lemmas,

we do not use the weak semantics without roll-back as a bridge between the strong semantics and the weak semantics with roll-back. We prefer to use the strong semantics with roll-back in that role.

### 6.3.1 Strong Semantics with Roll-back Implements Strong Semantics

**Lemma 6.1** *If  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', e, f, l, P \rangle$ , where the first transition is an instance of  $(\text{Trans Activate})_{rs}$  and subsequent ones are not unprotected transitions, nor instances of  $(\text{Trans Activate})_{rs}$  or  $(\text{Trans Block false } \dots)_{rs}$ , then  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma', T'.P, e \rangle$ .*

**Proof:** The proof is by induction on the length of  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', e, f, l, P \rangle$ . Applications of  $(\text{Trans Done})_{rs}$  in  $\mapsto_{rs}$  correspond to no transition with  $\mapsto_s$ . In all other cases, each applicable transition rule for  $\mapsto_{rs}^*$  has a corresponding one for  $\mapsto_s^*$ . ■

**Lemma 6.2** *If  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$ , where the first transition is an instance of  $(\text{Trans Activate})_{rs}$  and subsequent ones are not unprotected transitions nor instances of  $(\text{Trans Activate})_{rs}$ , and the computation includes instances of  $(\text{Trans Block false } \dots)_{rs}$ , then  $T = T'$  up to reordering and  $\sigma'$  extends  $\sigma$ .*

**Proof:** The computation must end with a sequence of applications of  $(\text{Trans Block false } \dots)_{rs}$  that undo any updates to  $\sigma$  (possibly followed by a sequence of applications of  $(\text{Trans Done})_{rs}$  to  $\langle \sigma', T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$ , with no effect). The first step takes an expression from  $T$ , while the application of  $\text{Trans Block false Restore})_{rs}$  returns it. ■

**Lemma 6.3** *If  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$ , where the first transition is an instance of  $(\text{Trans Activate})_{rs}$  and subsequent ones are not unprotected transitions, nor instances of  $(\text{Trans Activate})_{rs}$ , then there exist  $\sigma''$  and  $T''$  such that  $\sigma'$  extends  $\sigma''$ ,  $T'' = T'$  up to reordering, and  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \text{unit} \rangle$ .*

**Proof:** Lemma 6.1 deals with the case in which there are no occurrences of  $(\text{Trans Block false } \dots)_{rs}$  in the computation. Lemma 6.2 deals with the case in which there are such occurrences. ■

**Lemma 6.4** *If  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$ , then there exists  $\sigma''$  and  $T''$  such that  $\sigma'$  extends  $\sigma''$ ,  $T'' = T'$  up to reordering, and  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \text{unit} \rangle$ .*

**Proof:** The proof is by induction on the length of  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$ . We distinguish three cases:

- Suppose that the computation consists of a single transition

$$\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs} \langle \sigma', T', \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$$

This transition could be an unprotected transition, a trivial instance of  $(\text{Trans Done})_{rs}$ , or a trivial instance of  $(\text{Trans Activate})_{rs}$ . In all cases, we immediately have  $\langle \sigma, T, \mathbf{unit} \rangle = \langle \sigma', T', \mathbf{unit} \rangle$  or  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s \langle \sigma', T', \mathbf{unit} \rangle$ .

- If there is any intermediate state of the form  $\langle \sigma^*, T^*, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$ , then we conclude by applying the induction hypothesis to

$$\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma^*, T^*, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$$

and to

$$\langle \sigma^*, T^*, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$$

- Otherwise, it must be that the first step in the computation is an instance of  $(\text{Trans Activate})_{rs}$ , which is the only rule that can go from a state of the form  $\langle \sigma'', T'', \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$  to a state not of this same form. After this initial transition, unprotected transitions and  $(\text{Trans Activate})_{rs}$  are never enabled, because of the form of the intermediate states. We conclude by Lemma 6.3.

■

We extend the definition of violations to the states and computations of the strong semantics with roll-back, straightforwardly. For this purpose, given a state  $\langle \sigma, T, e, f, l, P \rangle$ , we consider only its first three components,  $\langle \sigma, T, e \rangle$ , and apply the definitions above.

**Lemma 6.5** *Assume that all strong computations that start from the state  $\langle \sigma, T, \mathbf{unit} \rangle$  are violation-free. Then all strong computations with roll-back that start from the state  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$  are violation-free.*

**Proof:** We argue by contradiction. Suppose that a state with a violation was reachable:  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', e', f', l', P' \rangle$ . We exclude any final occurrences of instances of  $(\text{Trans Block false } \dots)_{rs}$  in this computation, since they cannot introduce any violations not already present. We

show that  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma'', T'', e' \rangle$  where  $\sigma'$  extends  $\sigma''$  and  $T'' = T'.P'$  up to reordering. A violation in  $\langle \sigma', T', e', f', l', P' \rangle$  immediately implies a violation in  $\langle \sigma'', T'', e' \rangle$ .

Lemma 6.4 enables us to simulate any prefixes  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \cdot, \cdot, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$  of  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', e', f', l', P' \rangle$ . After the longest such prefix, the first transition is an instance of  $(\text{Trans Activate})_{rs}$  and subsequent ones are not unprotected transitions, nor instances of  $(\text{Trans Activate})_{rs}$  or  $(\text{Trans Block false } \dots)_{rs}$ . Lemma 6.1 yields that  $\langle \sigma, T, \text{unit} \rangle \mapsto_s^* \langle \sigma'', T'', e' \rangle$  where  $\sigma'$  extends  $\sigma''$  and  $T'' = T'.P'$  up to reordering. ■

### 6.3.2 Weak Semantics with Roll-back Implements Strong Semantics with Roll-back

**Lemma 6.6** *Suppose that  $\langle \sigma, T, e, f, l, P \rangle \mapsto_{rw} \langle \sigma', T', e', f', l', P' \rangle \mapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$  where the first transition is an instance of  $(\text{Trans Activate})_{rw}$  or a protected transition and the second transition is an unprotected transition. Suppose further that  $\langle \sigma, T, e, f, l, P \rangle$  is violation-free, and that, if  $r \in \text{dom}(l)$ , then no element of  $T$  is of the form  $\mathcal{U}[\ !r \ ]$  or  $\mathcal{U}[ r := \dots ]$ . Then there exists  $\langle \sigma^*, T^*, e^*, f^*, l^*, P^* \rangle$  such that  $\langle \sigma, T, e, f, l, P \rangle \mapsto_{rw} \langle \sigma^*, T^*, e^*, f^*, l^*, P^* \rangle \mapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$  where the second transition is an instance of  $(\text{Trans Activate})_{rw}$  or a protected transition.*

**Proof:** The proof is analogous to that of Lemma 5.2.

Consider a transition that is an instance of  $(\text{Trans Activate})_{rw}$  followed by an unprotected transition. As in Lemma 5.2, we can rewrite this pair of transitions so that the unprotected transition is before the instance of  $(\text{Trans Activate})_{rw}$ . The unprotected transition uses the same rule as before the commutation, with only a straightforward change in the choice of evaluation context.

Consider a protected transition followed by an unprotected transition. As in Lemma 5.2, we can rewrite this pair of transitions so that the unprotected transition is before the protected transition, with an intermediate state constructed by case analysis on the protected transition. Most of the cases of protected transitions are as the corresponding cases in Lemma 5.2.

- In the cases of  $(\text{Trans Appl P})_{rw}$ ,  $(\text{Trans Block true P})_{rw}$ ,  $(\text{Trans Ref P})_{rw}$ , and  $(\text{Trans Deref P})_{rw}$ , the state components  $f$ ,  $l$ , and  $P$  are constant through the transitions, so we proceed exactly as in Lemma 5.2, letting  $f^* = f$ ,  $l^* = l$ , and  $P^* = P$ .

- The case of  $(\text{Trans Async P})_{rw}$  is similar except that  $P$  varies, and we let  $P^* = P$ .
- The case of  $(\text{Trans Set P})_{rw}$  is also similar except that  $l$  varies, and we let  $l^* = l$ .
- The case of  $(\text{Trans Block false Undo})_{rw}$  is analogous to that of  $(\text{Trans Set P})_{rw}$ , and we let  $l^* = l$  again. We use the hypothesis that if  $r \in \text{dom}(l)$ , then no element of  $T$  is of the form  $\mathcal{U}[!r]$  or  $\mathcal{U}[r := \dots]$ , instead of the hypothesis that the state  $\langle \sigma, T, e, f, l, P \rangle$  is violation-free.

■

**Lemma 6.7** *Assume that all strong computations with roll-back that start from the state  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$  are violation-free. If  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \xrightarrow{rs}^* \langle \sigma', T', e', f', l', P' \rangle$  then, if  $r \in \text{dom}(l')$ , then no element of  $T'$  is of the form  $\mathcal{U}[!r]$  or  $\mathcal{U}[r := \dots]$ .*

**Proof:** We show that if  $r \in \text{dom}(l')$ , then no element of  $T'$  is of the form  $\mathcal{U}[!r]$  or  $\mathcal{U}[r := \dots]$ . We proceed by induction on the length of the computation, and argue by cases on the last step of the computation. If  $l' = \emptyset$ , then the claim is trivially true, so it suffices to consider the cases of rules  $(\text{Trans } \dots P)_{rs}$  and  $(\text{Trans Block false Undo})_{rs}$ . In all cases except that of  $(\text{Trans Set P})_{rs}$ , we have that  $l'$  is equal or shorter than its previous value while  $T'$  equals its previous value, so the induction hypothesis yields the desired result. In the case of  $(\text{Trans Set P})_{rs}$ , since the previous state is violation-free and its active expression is a write to the location  $r$  being added to  $l'$ , the expressions in  $T'$  cannot be accessing this location. ■

**Lemma 6.8** *Assume that all strong computations with roll-back that start from the state  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle$  are violation-free. Consider a weak computation with roll-back  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \xrightarrow{rw}^* \langle \sigma', T', e', f', l', P' \rangle$ . Then there is a strong computation with roll-back  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \xrightarrow{rs}^* \langle \sigma', T', e', f', l', P' \rangle$ .*

**Proof:** Consider a weak computation  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \xrightarrow{rw}^* \langle \sigma', T', e', f', l', P' \rangle$ . Much as in Theorem 5.3, the proof is by induction on the length of this computation, relying on Lemma 6.6. As in Theorem 5.3, we strengthen the claim with the assertion that the strong computation is of the same length as the original computation.

The claim is vacuously true when the length of the computation  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \xrightarrow{rw}^* \langle \sigma', T', e', f', l', P' \rangle$  is 0. For the inductive step, suppose that we have  $\langle \sigma, T, \text{unit}, \text{unit}, \emptyset, \emptyset \rangle \xrightarrow{rw}^* \langle \sigma', T', e', f', l', P' \rangle$ .

$\langle \sigma', T', e', f', l', P' \rangle \mapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ . By induction hypothesis,  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', e', f', l', P' \rangle \mapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ . If  $\langle \sigma', T', e', f', l', P' \rangle \mapsto_{rs} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ , then we immediately obtain a strong computation  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ . On the other hand, if  $\langle \sigma', T', e', f', l', P' \rangle \mapsto_{rw} \langle \sigma'', T'', e'', f'', l'', P'' \rangle$  is not strong, then there must be a transition before it (since otherwise  $e' = e = f' = f = \mathbf{unit}$  and  $l' = l = P' = P = \emptyset$ ), in other words a last transition in  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma', T', e', f', l', P' \rangle$ . Moreover this strong transition to  $\langle \sigma', T', e', f', l', P' \rangle$  cannot be an instance of  $(\text{Trans Unprotect})_{rs}$ ,  $(\text{Trans Block false Restore})_{rs}$ , or  $(\text{Trans Done})_{rs}$  nor an unprotected strong transition, for otherwise  $e' = f' = \mathbf{unit}$  and  $l' = P' = \emptyset$ , so it must be an instance of  $(\text{Trans Activate})_{rs}$  or a protected transition. We can apply Lemma 6.6 because of Lemma 6.7. By Lemma 6.6, we can commute steps in order to obtain a strong computation  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ . Specifically, Lemma 6.6 shows how to move an unprotected transition before an instance of  $(\text{Trans Activate})_{rs}$  or a protected transition. By induction hypothesis, the computation up to this unprotected transition can be made strong, and adding back the instance of  $(\text{Trans Activate})_{rs}$  or the protected transition yields the desired strong computation  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rs}^* \langle \sigma'', T'', e'', f'', l'', P'' \rangle$ . ■

### 6.3.3 Weak Semantics with Roll-back Implements Strong Semantics

We obtain the following theorem, which enables us to use the strong semantics in reasoning about systems without violations, here with roll-backs. Unlike Theorem 5.3, this theorem is restricted to computations that lead to states of a particular form, in particular with an active expression  $\mathbf{unit}$ . In general, when the active expression is not  $\mathbf{unit}$ , the intermediate store  $\sigma'$  may be one that cannot be obtained by strong computations. Moreover, unlike in Theorem 5.3, this theorem does not yield a strong computation with exactly the same final store: intuitively, the computation with roll-backs may allocate additional locations, and those are not de-allocated. However, the two final stores coincide at all accessible locations: our invariant on states imply that both stores are defined (and equal) at all referenced locations.

**Theorem 6.9** *Assume that all strong computations that start from the state  $\langle \sigma, T, \mathbf{unit} \rangle$  are violation-free. Consider a weak computation  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rw}^* \langle \sigma', T', \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$ , possibly with roll-backs. Then there is a strong com-*

putation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \mathbf{unit} \rangle$  for some  $\sigma''$  and  $T''$  such that  $\sigma'$  is an extension of  $\sigma''$  and  $T'' = T'$  up to reordering.

**Proof:** Lemma 6.5 implies that all strong computations with roll-back that start from the state  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$  are violation-free, so Lemma 6.8 is applicable. The theorem follows from the composition of Lemmas 6.4 and 6.8. ■

Much as in Section 5, we deduce a correctness theorem with a static check as hypothesis:

**Corollary 6.10** *Assume that  $\langle \sigma, T, \mathbf{unit} \rangle$  is well-typed. Consider a weak computation  $\langle \sigma, T, \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle \mapsto_{rw}^* \langle \sigma', T', \mathbf{unit}, \mathbf{unit}, \emptyset, \emptyset \rangle$ , possibly with roll-backs. Then there is a strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \mathbf{unit} \rangle$  for some  $\sigma''$  and  $T''$  such that  $\sigma'$  is an extension of  $\sigma''$  and  $T'' = T'$  up to reordering.*

**Proof:** This is an immediate consequence of Corollary 5.1 and Theorem 6.9. ■

## 7 Optimistic Concurrency

In our high-level semantics, we do not model the workings of optimistic concurrency. Here we treat an extension of the operational semantics in which multiple active expressions are evaluated simultaneously, with roll-backs in case of conflict. The semantics includes in-place updates to the reference store, thus modeling an efficient but problematic technique.

### 7.1 An Informal Account

As with roll-backs, an implementation of our language that includes optimistic concurrency may not respect the semantics for all programs. Indeed, in semantics where unprotected transitions can be interleaved with protected transitions arbitrarily, the unprotected transitions may be able to see intermediate states with conflicts that result from optimistic concurrency.

Even the absence of high-level violations is not enough in this case. Unless one adopts somewhat inefficient semantics (e.g., with buffered writes), it is possible that a program has no violations in the strong semantics but has violations in lower-level semantics with optimistic concurrency. Basically, optimistic concurrency can give rise to transient states with violations that have no high-level counterpart.

The following small program from Section 4 illustrates this point:

```
let y = ref true in
  async (y := false;
         unprotected z := !y);
  async (blockUntil !y;
         y := true)
```

According to the strong semantics, the program can set  $z$  only to `false`. In an implementation with optimistic concurrency, the two fragments `y := false; unprotected z := !y` and `blockUntil !y; y := true` may be executed simultaneously, and a conflict may be detected. Specifically, such an implementation may run `blockUntil !y`, then `y := false`, then `y := true`, then `unprotected z := !y`, and finally it may roll back one or both fragments once a conflict is detected, but the roll-back may not include resetting  $z$ , so the implementation may set  $z$  to `true`. This sort of surprising behavior is a so-called privatization problem. It affects some programs that rely on common idioms, and which are not nearly as artificial as our small example, such as:

```
let x = ref V in
  let y = ref true in
  async (y := false;
         unprotected z := !x);
  async (blockUntil !y;
         x := V')
```

where  $V$  and  $V'$  are distinct values. Here, intuitively, the contents of the reference location  $y$  indicates whether  $x$  is shared. Setting that location to `false` amounts to a privatization. In the first of these two examples, although a violation is not apparent at the source level, an optimistic scheme may conceivably be ready to execute `y := true` and `z := !y` at the same time. In the second, similarly, an optimistic scheme may conceivably be ready to execute `x := V'` and `z := !x` at the same time.

Putting a condition directly on the lower-level semantics may be adequate as a basis for compiler optimizations but is not fully satisfactory—programmers should not be aware of the details of this semantics, and it would be a slippery slope to making a condition that immediately guarantees correctness. Instead, we use the type system for separation at the source level.

As noted above, we treat an extension of the operational semantics in which multiple active expressions are evaluated simultaneously, with roll-backs in case of conflict. This semantics is a little simplistic in comparison

---

**State space**

$$\begin{aligned} S &\in \text{State} = \text{RefStore} \times \text{ExpSeq} \times \text{TrySeq} \times \text{Log} \\ \sigma &\in \text{RefStore} = \text{RefLoc} \rightarrow \text{Value} \\ l &\in \text{Log} = (\text{RefLoc} \times \text{Value})^* \\ r &\in \text{RefLoc} \subset \text{Var} \\ T, P &\in \text{ExpSeq} = \text{Exp}^* \\ O &\in \text{TrySeq} = \text{Try}^* \\ d &\in \text{Try} = \text{Exp} \times \text{Exp} \times \text{Accesses} \times \text{ExpSeq} \\ a &\in \text{Accesses} = \text{RefLoc}^* \end{aligned}$$

Figure 14: State space, with optimistic concurrency.

---

with actual implementations, partly in order to avoid tedious book-keeping that would obscure the presentation. On the other hand, this semantics includes in-place updates, which are attractively efficient but which can contribute to semantic problems, as in the examples above. Buffering updates could perhaps avoid these problems, but would also result in run-time costs. We believe that a semantics with buffering would lead to easier but less interesting results, hence our choice.

## 7.2 A Lower-Level Semantics with Optimistic Concurrency

We give weak lower-level semantics with optimistic concurrency. (Strong lower-level semantics does not make as much sense in this case, and does not seem to help as much: the most straightforward strong semantics with optimistic concurrency does not implement the strong semantics. So we do not give a strong semantics.)

### 7.2.1 States

As described in Figure 14, states become more complex. In addition to the components  $\sigma$ ,  $T$ , and  $l$  that appear in the semantics with roll-back, there is here a list of tuples instead of two simple active expressions  $e$  and its origin expression  $f$ . Each of the tuples is called a try. A try consists of the following components:

- an active expression  $e$ ,
- its origin expression  $f$ , as in the semantics with roll-back,

- a description of the accesses that  $e$  has performed, which are used for conflict detection and which here is simply a list of reference locations,
- a list  $P$  of threads to be forked upon commit.

Clearly these components could be refined further in more elaborate schemes. For instance, conflict detection could distinguish reads and writes, possibly with timestamps; moreover, the log used for undos could contain additional information in order to support more selective undos. Still, even in the present form, the semantics exhibits interesting and challenging features.

### 7.2.2 Steps

The rules of the semantics are given in Figure 15. They rely on the following definitions:

- We say that  $(e_i, f_i, a_i, P_i)$  and  $(e_j, f_j, a_j, P_j)$  conflict if  $a_i$  and  $a_j$  have at least one element in common.
- We say that  $(e, f, a, P)$  conflicts with  $O$  if  $(e, f, a, P)$  conflicts with some tuple in  $O$ .
- We say that  $O$  conflicts if it contains two distinct tuples that conflict.
- Given a log  $l$  and a list of reference locations  $a$ , we write  $l - a$  for the log obtained from  $l$  by restricting to reference locations not in  $a$ .
- If  $O$  is  $(e_1, f_1, a_1, P_1) \cdots (e_n, f_n, a_n, P_n)$  then  $origin(O)$  is the list  $f_1 \cdots f_n$ .
- $\sigma l$  is the result of applying all elements of  $l$  to  $\sigma$ .

We allow for conflicts to be detected as soon as they occur, but we do not require it. For simplicity, we do not include some secondary features sufficiently explored in the semantics with roll-back. In particular we do not model undoing updates one by one—here, all the updates recorded in a log may be undone at once. Also, we do not give a special treatment for `blockUntil false`; we simply allow undo to happen at any point (possibly because of conflicts, but also possibly because `blockUntil false`).

As above, we distinguish protected and unprotected transitions. In this semantics, each transition has the form

$$\langle \sigma, T, O, l \rangle \mapsto_{ow} \langle \sigma', T', O', l' \rangle$$

---

**Transition rules**

$\langle \sigma, T, O.(\mathcal{P}[\lambda x. e] V], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[e[V/x]], f, a, P).O', l \rangle$	(Trans Appl P) <sub>ow</sub>
$\langle \sigma, T, \mathcal{U}[\lambda x. e] V].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, \mathcal{U}[e[V/x]].T', O, l \rangle$	(Trans Appl U) <sub>ow</sub>
$\langle \sigma, T, O.(\mathcal{P}[\text{ref } V], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma[r \mapsto V], T, O.(\mathcal{P}[r], f, a, P).O', l \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	(Trans Ref P) <sub>ow</sub>
$\langle \sigma, T, \mathcal{U}[\text{ref } V].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma[r \mapsto V], T, \mathcal{U}[r].T', O, l \rangle$ if $r \in \text{RefLoc} - \text{dom}(\sigma)$	(Trans Ref U) <sub>ow</sub>
$\langle \sigma, T, O.(\mathcal{P}[\! r], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[r], f, r.a, P).O', l \rangle$ if $\sigma(r) = V$	(Trans Deref P) <sub>ow</sub>
$\langle \sigma, T, \mathcal{U}[\! r].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, \mathcal{U}[V].T', O, l \rangle$ if $\sigma(r) = V$	(Trans Deref U) <sub>ow</sub>
$\langle \sigma, T, O.(\mathcal{P}[r := V], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma[r \mapsto V], T, O.(\mathcal{P}[\text{unit}], f, r.a, P).O', l' \rangle$ where $l' = \text{if } r \in \text{dom}(l) \text{ then } l \text{ else } l.[r \mapsto \sigma(r)]$	(Trans Set P) <sub>ow</sub>
$\langle \sigma, T, \mathcal{U}[r := V].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma[r \mapsto V], T, \mathcal{U}[\text{unit}].T', O, l \rangle$	(Trans Set U) <sub>ow</sub>
$\langle \sigma, T, O.(\mathcal{P}[\text{async } e], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[\text{unit}], f, a, e.P).O', l \rangle$	(Trans Async P) <sub>ow</sub>
$\langle \sigma, T, \mathcal{U}[\text{async } e].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, e.T, \mathcal{U}[\text{unit}].T', O, l \rangle$	(Trans Async U) <sub>ow</sub>
$\langle \sigma, T, O.(\mathcal{P}[\text{blockUntil true}], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, O.(\mathcal{P}[\text{unit}], f, a, P).O', l \rangle$	(Trans Block P) <sub>ow</sub>
$\langle \sigma, T, \mathcal{U}[\text{blockUntil true}].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, \mathcal{U}[\text{unit}].T', O, l \rangle$	(Trans Block U) <sub>ow</sub>
$\langle \sigma, T, O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, l, \text{origin}(O).T, \emptyset, \emptyset \rangle$	(Trans Undo) <sub>ow</sub>
$\langle \sigma, T, O.(\mathcal{P}[\text{unprotected } e], f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, \mathcal{P}[\text{unprotected } e].P, O.O', l - a \rangle$ if $(\mathcal{P}[\text{unprotected } e], f, a, P)$ does not conflict with $O.O'$	(Trans Unprotect) <sub>ow</sub>
$\langle \sigma, T, O.(\text{unit}, f, a, P).O', l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, P, O.O', l - a \rangle$ if $(\text{unit}, f, a, P)$ does not conflict with $O.O'$	(Trans Done) <sub>ow</sub>
$\langle \sigma, T, \mathcal{E}[\text{unprotected } V].T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, \mathcal{E}[V].T', O, l \rangle$	(Trans Close) <sub>ow</sub>
$\langle \sigma, T, e.T', O, l \rangle$	$\mapsto_{ow}$	$\langle \sigma, T, T', (e, e, \emptyset, \emptyset).O, l \rangle$	(Trans Activate) <sub>ow</sub>

Figure 15: Transition rules of the abstract machine, with optimistic concurrency (weak).

---

In many cases, a transition is defined in terms of a context that has a hole either in  $T$  and in  $T'$ , or in  $O$  and in  $O'$ . We say that the transition is protected if the hole is in  $O$  and in  $O'$ , and say that the transition is unprotected if the hole is in  $T$  and in  $T'$ . By definition, we have:

- transitions that are instances of  $(\text{Trans } \dots P)_{ow}$  are always protected;
- transitions that are instances of  $(\text{Trans } \dots U)_{ow}$  or of  $(\text{Trans Close})_{ow}$  are always unprotected;
- transitions that are instances of  $(\text{Trans Undo})_{ow}$ ,  $(\text{Trans Unprotect})_{ow}$ ,  $(\text{Trans Done})_{ow}$ , or  $(\text{Trans Activate})_{ow}$  are neither protected nor unprotected.

### 7.3 Soundness of the Type System for Separation, Revisited

This proof is essentially a refinement of that of Section 4.2.

#### 7.3.1 Typing States

We extend the type system to states  $\langle \sigma, T, O, l \rangle$ . We write:

$$E \vdash \langle \sigma, T, O, l \rangle$$

if

- $\text{dom}(\sigma) = \text{dom}(E) \cap \text{RefLoc}$ ,
- for all  $r \in \text{dom}(\sigma)$ , there exist  $t$  and  $p$  such that  $E(r) = \text{Ref}_p t$  and  $E; p \vdash \sigma(r) : t$ ,
- for each  $e'$  in  $T$ ,  $E; P \vdash e' : \text{Unit}$ ,
- for each  $(e, f, a, P)$  in  $O$ ,  $E; P \vdash e : \text{Unit}$  and  $E; P \vdash f : \text{Unit}$ , and for each  $e'$  in  $P$ ,  $E; P \vdash e' : \text{Unit}$ ,
- for each  $r \in \text{dom}(l)$ , there exists  $t$  such that  $E(r) = \text{Ref}_p t$  and  $E; P \vdash l(r) : t$ .

In the special case where  $O$  and  $l$  are empty, we may omit them and simply say that  $\langle \sigma, T \rangle$  is well-typed. Note that this is equivalent to  $\langle \sigma, T, \text{unit} \rangle$  being well-typed according to the definition of Section 4.2. Thus, whether  $\langle \sigma, T \rangle$  is well-typed can be understood and proved entirely in terms of the higher-level definitions, without any regard for optimistic concurrency.

### 7.3.2 Auxiliary Results

Lemmas 4.1, 4.2, and 4.3 continue to hold, as they do not concern the operational semantics. The following are slight, trivial adaptations of Lemmas 4.4 and 4.5 that take into account the form of states used in this section.

**Lemma 7.1** *Assume that  $r \in \text{dom}(\sigma)$  and  $E(r) = \text{Ref}_{p_0} t_0$ . If  $E \vdash \langle \sigma, T, O, l \rangle$  and  $E; p_0 \vdash V : t_0$ , then  $E \vdash \langle \sigma[r \mapsto V], T, O, l \rangle$ .*

**Lemma 7.2** *If  $E \vdash \langle \sigma, T, O.(\mathcal{P}[e], f, a, P).O', l \rangle$  then there exists  $t$  such that  $E; P \vdash e : t$ . If  $E \vdash \langle \sigma, T\mathcal{U}[e'], T', O, l \rangle$  then there exists  $t$  such that  $E; U \vdash e' : t$ .*

### 7.3.3 Computation Preserves Typability

We obtain that typability is preserved by computation.

**Theorem 7.3 (Preservation of Typability)** *Suppose that  $\langle \sigma, T, O, l \rangle \xrightarrow{*}_{ow} \langle \sigma', T', O', l' \rangle$ . If  $\langle \sigma, T, O, l \rangle$  is well-typed, then so is  $\langle \sigma', T', O', l' \rangle$ .*

**Proof:** The proof is an extension of that of Theorem 4.6.

The cases of  $(\text{Trans Appl P})_{ow}$ ,  $(\text{Trans Appl U})_{ow}$ ,  $(\text{Trans Ref P})_{ow}$ ,  $(\text{Trans Ref U})_{ow}$ ,  $(\text{Trans Deref P})_{ow}$ ,  $(\text{Trans Deref U})_{ow}$ ,  $(\text{Trans Set U})_{ow}$ ,  $(\text{Trans Async U})_{ow}$ ,  $(\text{Trans Block P})_{ow}$ ,  $(\text{Trans Block U})_{ow}$ , and  $(\text{Trans Close})_{ow}$  are like the corresponding cases in that proof.

In the case of  $(\text{Trans Set P})_{ow}$ , we also have that the updated  $l$  is well-typed, and the location added to  $l$  has a type of the form  $\text{Ref}_P t_0$  by Lemma 7.2, which gives the effect  $P$  to the assignment. In the case of  $(\text{Trans Async P})_{ow}$ , similarly, we also have that the updated  $P$  is well-typed, straightforwardly.

The cases of  $(\text{Trans Done})_{ow}$  and  $(\text{Trans Unprotect})_{ow}$  are directly analogous to that of  $(\text{Trans Unprotect})_s$ .

In the case of  $(\text{Trans Undo})_{ow}$ , the updates to the store preserve typing by Lemma 7.1, and the origin expressions (which are added back to the pool) are well-typed by hypothesis.

The case of  $(\text{Trans Activate})_{ow}$  is straightforward because the new active expression and its origin all come from the pool, which is well-typed by hypothesis, and  $l$  and  $P$  are empty in this case. ■

Examining the proof again, we note that if  $\langle \sigma, T, O, l \rangle$  is well-typed with respect to an environment  $E$ , then  $\langle \sigma', T', O', l' \rangle$  is well-typed with respect

to an extension of  $E$ . In the cases of  $(\text{Trans Ref } \dots)_{ow}$ ,  $(\text{Trans Deref } \dots)_{ow}$ , and  $(\text{Trans Set } \dots)_{ow}$ , which deal with a reference location of type  $\mathbf{Ref}_{p_0} t_0$ , we also note that if the transition is protected, then  $p_0$  must be  $\mathbf{P}$ , and if the transition is unprotected, then  $p_0$  must be  $\mathbf{U}$ , by Lemma 7.2. It follows that, if  $\langle \sigma, T, O, l \rangle \mapsto_{ow}^* \langle \sigma', T', O', l' \rangle$  and  $\langle \sigma, T, O, l \rangle$  is well-typed, then there exist subsets  $\mathbf{P}$  and  $\mathbf{U}$  of  $\text{dom}(\sigma')$  such that the protected transitions in  $\langle \sigma, T, O, l \rangle \mapsto_{ow}^* \langle \sigma', T', O', l' \rangle$  allocate, read, or write only reference locations in  $\mathbf{P}$ , and the unprotected transitions in  $\langle \sigma, T, O, l \rangle \mapsto_{ow}^* \langle \sigma', T', O', l' \rangle$  allocate, read, or write only reference locations in  $\mathbf{U}$ . Moreover, reference locations reset by  $(\text{Trans Undo})_{ow}$  are in  $\mathbf{P}$ . The subsets in question consist of the reference locations declared with effects  $\mathbf{P}$  and  $\mathbf{U}$ , respectively, in the environment.

## 7.4 Correctness

We prove the correctness for the weak semantics with optimistic concurrency with respect to the strong semantics (without optimistic concurrency).

**Theorem 7.4** *Assume that  $\langle \sigma, T \rangle$  is well-typed. Consider a computation  $\langle \sigma, T, \emptyset, \emptyset \rangle \mapsto_{ow}^* \langle \sigma', T', \emptyset, \emptyset \rangle$ . Then there is a strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \mathbf{unit} \rangle$  for some  $\sigma''$  and  $T''$  such that  $\sigma'$  is an extension of  $\sigma''$  and  $T'' = T'$  up to reordering.*

**Proof:** Assuming that  $\langle \sigma, T \rangle$  is well-typed, more generally we consider a computation  $\langle \sigma, T, \emptyset, \emptyset \rangle \mapsto_{ow}^* \langle \sigma', T', O', l' \rangle$ , and we prove that:

1. There is a strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \mathbf{unit} \rangle$  where:
  - $\sigma' l'$  is an extension of  $\sigma''$ ,
  - $T'' = T'.\text{origin}(O')$  up to reordering.
2. Moreover, if  $O' = O^\dagger.(e, f, a, P).O^{\dagger\dagger}$  and  $(e, f, a, P)$  does not conflict with  $O^\dagger.O^{\dagger\dagger}$ , then there is a further strong computation  $\langle \sigma'', T'', \mathbf{unit} \rangle \mapsto_s \langle \sigma'', T''', f \rangle \mapsto_s^* \langle \sigma''', T'''.P, e \rangle$  where the first transition is an instance of  $(\text{Trans Activate})_s$  and
  - $\sigma'(l' - a)$  is an extension of  $\sigma'''$ ,
  - $T''' = T'.\text{origin}(O^\dagger.O^{\dagger\dagger})$  up to reordering.

The proof is by induction on the computation  $\langle \sigma, T, \emptyset, \emptyset \rangle \mapsto_{ow}^* \langle \sigma', T', O', l' \rangle$ , with a case analysis on the last rule applied. In most cases, the construction of a suitable strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \mapsto_s^* \langle \sigma'', T'', \mathbf{unit} \rangle$  is straightforward using the induction hypothesis and typing. The cases of  $(\text{Trans Unprotect})_{ow}$  and  $(\text{Trans Done})_{ow}$  rely on the further strong computation of claim (2). Throughout, we work up to reorderings in the pool. We use the sets  $\mathsf{P}$  and  $\mathsf{U}$  as determined by  $\langle \sigma, T, \emptyset, \emptyset \rangle \mapsto_{ow}^* \langle \sigma', T', O', l' \rangle$  and Theorem 7.3.

- $(\text{Trans Appl P})_{ow}$ ,  $(\text{Trans Deref P})_{ow}$ ,  $(\text{Trans Async P})_{ow}$ ,  $(\text{Trans Block P})_{ow}$ :
  1. The desired strong computation is that given by the induction hypothesis, since  $\text{origin}(O')$ ,  $\sigma'$ , and  $l'$  do not change in these cases.
  2. The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an extension by a corresponding application of  $(\text{Trans Appl})_s$ ,  $(\text{Trans Deref})_s$ ,  $(\text{Trans Async})_s$ , or  $(\text{Trans Block})_s$ , respectively (for the try where the transition operates).
- $(\text{Trans Ref P})_{ow}$ :
  1. The desired strong computation is that given by the induction hypothesis, since  $\text{origin}(O')$  does not change,  $\sigma'$  is extended with a new location, and  $l'$  does not change in this case.
  2. The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an extension by a corresponding transition (for the try where the transition operates). In the latter case, we use that the reference location being allocated is not in the domain of  $l' - a$ .
- $(\text{Trans Set P})_{ow}$ :
  1. The desired strong computation is that given by the induction hypothesis, since  $\text{origin}(O')$  does not change, and since  $\sigma'l'$  remains the same even if  $\sigma'$  and  $l'$  may change.
  2. The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an extension by a corresponding transition (for the try where the transition operates). In the former case, the absence of conflict

implies the required commutation: in case the instance of  $(\text{Trans Set P})_{ow}$  adds a reference location in  $l'$ , this reference location is not in  $a$ , so  $\sigma'(l' - a) = \sigma^*(l^* - a)$ , where  $\sigma^*$  and  $l^*$  are the reference store and the log before the transition. In the latter case, we use that  $l' - a = l^* - a^*$ , where  $l^*$  and  $a^*$  are the reference store and the list of accessed addresses of the try before the transition, and that the reference location being set is not in the domain of  $l' - a$ .

- $(\text{Trans Appl U})_{ow}$ ,  $(\text{Trans Deref U})_{ow}$ ,  $(\text{Trans Async U})_{ow}$ ,  $(\text{Trans Block U})_{ow}$ ,  $(\text{Trans Close})_{ow}$ :
  1. The induction hypothesis gives a strong computation that we extend with a corresponding application of  $(\text{Trans Appl})_s$ ,  $(\text{Trans Deref})_s$ ,  $(\text{Trans Async})_s$ ,  $(\text{Trans Block})_s$ , or  $(\text{Trans Close})_s$ , respectively.
  2. The further strong computation is that given by the induction hypothesis, with straightforward changes in the pool that correspond to the application of  $(\text{Trans Appl})_s$ ,  $(\text{Trans Deref})_s$ ,  $(\text{Trans Async})_s$ ,  $(\text{Trans Block})_s$ , or  $(\text{Trans Close})_s$ , respectively.
- $(\text{Trans Ref U})_{ow}$ :
  1. The induction hypothesis gives a strong computation that we extend with a corresponding application of  $(\text{Trans Ref})_s$ . Typing (and Theorem 7.3) imply that the application of the log  $l'$  and the allocation commute as they are to different locations, since the domain of a log consists of reference locations in  $P$ .
  2. The further strong computation is basically that given by the induction hypothesis, up to an extension in  $U$ , since a try can depend only on reference locations in  $P$ , and typing implies that those cannot be allocated by  $(\text{Trans Ref U})_{ow}$ .
- $(\text{Trans Set U})_{ow}$ :
  1. The induction hypothesis gives a strong computation that we extend with a corresponding application of  $(\text{Trans Set})_s$ . Typing (and Theorem 7.3) imply that the application of the log  $l'$  and the update commute as they are to different locations, since the domain of a log consists of reference locations in  $P$ .

2. The further strong computation is basically that given by the induction hypothesis, up to a change to the initial value of a reference location in  $\mathbf{U}$ , since a try can depend only on reference locations in  $\mathbf{P}$ , and typing implies that those cannot be updated by  $(\text{Trans Set U})_{ow}$ .
- $(\text{Trans Activate})_{ow}$ :
    1. This case is also by a straightforward application of the induction hypothesis. The strong computation is that given by the induction hypothesis, since  $l'$  does not change, and since the origin of the new try comes from the pool.
    2. The further strong computation is either that given by the induction hypothesis (if the transition belongs to a different try) or an instance of  $(\text{Trans Activate})_s$  (for the try that the transition generates).
  - $(\text{Trans Unprotect})_{ow}$ :
    1. Suppose that  $\langle \sigma, T, \emptyset, \emptyset \rangle \xrightarrow{ow}^* \langle \sigma', T', O', l' \rangle$  where the last rule applied is an instance of  $(\text{Trans Unprotect})_{ow}$ . If the state before the transition is  $\langle \sigma^*, T^*, O^*.(\mathcal{P}[\text{unprotected } e^*], f^*, a^*, P^*).O^{**}, l^* \rangle$ , then
      - $\sigma^* = \sigma'$ ,
      - $T' = T^*. \mathcal{P}[\text{unprotected } e^*].P^*$ ,
      - $O' = O^*.O^{**}$ , and
      - $l' = l^* - a^*$ .

Moreover,  $(\mathcal{P}[\text{unprotected } e^*], f^*, a^*, P^*)$  does not conflict with  $O^*.O^{**}$ , that is, with  $O'$ .

The induction hypothesis implies that there is a strong computation  $\langle \sigma, T, \mathbf{unit} \rangle \xrightarrow{s}^* \langle \sigma'', T^*.origin(O^*.O^{**}).f^*, \mathbf{unit} \rangle$  where  $\sigma'l^*$  is an extension of  $\sigma''$ . Moreover, the induction hypothesis implies that there is a further strong computation  $\langle \sigma'', T^*.origin(O^*.O^{**}).f^*, \mathbf{unit} \rangle \xrightarrow{s} \langle \sigma'', T^*.origin(O^*.O^{**}), f^* \rangle \xrightarrow{s}^* \langle \sigma'', T^*.origin(O^*.O^{**}).P^*, \mathcal{P}[\text{unprotected } e^*] \rangle$  where  $\sigma'(l^* - a^*)$  is an extension of  $\sigma''$ , that is, where  $\sigma'l'$  is an extension of  $\sigma''$ . We obtain the desired strong computation by applying  $(\text{Trans Unprotect})_s$ .
    2. The further strong computation is basically that given by the induction hypothesis, up to changes in the initial values of reference locations in  $a^*$ . The absence of conflict that is the hypothesis of

the application of  $(\text{Trans Unprotect})_{ow}$  implies that  $a$  and  $a^*$  do not intersect, so the initial values of reference locations in  $a^*$  do not affect a try that accesses only locations in  $a$ .

- $(\text{Trans Done})_{ow}$ : This case is almost identical to that of  $(\text{Trans Unprotect})_{ow}$  (but slightly simpler).

■