Fault-Tolerant Voting in a Simply-Typed Lambda Calculus

Martin Elsman
IT University of Copenhagen, Denmark
mael@itu.dk

Abstract

In this paper we present a translation from the simply typed lambda calculus into an extended simply typed lambda calculus that enables programs to be tolerant to transient faults. The translation triples all basic instructions and generates majority voting code for ensuring correct control-flow.

We demonstrate that the generated voting code causes evaluation to proceed successfully even in the presence of a transient fault corrupting a register other than a control flow register (i.e., the stack pointer or the instruction pointer).

Categories and Subject Descriptors D.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

Keywords Transient Faults, Lambda Calculus, Compilation, Optimization

1. Introduction

A transient fault occurs when an energetic ray or a particle cause a bit-flip in the central processing unit, causing a change in the content of a register or an error in the instruction logic causing the result register of an instruction to become semantically wrong.

Transient faults (also called soft errors) are arguably becoming a larger and larger problem for computing. When performance of microprocessors increase with a growing number of transistors per square millimeter, processors are also becoming more volatile to transient faults caused by various kinds of radiation (alpha particles, neutrons, and cosmic rays) [4, 14]. Much work has focused on using software-only techniques and hardware/software hybrid techniques for fault detection and fault tolerance for transient faults (see [14, 13] for some good overviews), but only recently have correctness properties of the techniques been investigated formally [18, 8].

In previous work, Walker et al. [18] have given a translation from the simply typed lambda calculus into λ_{zap}, an extended lambda calculus that uses majority voting to guarantee well-typed λ_{zap} programs to execute correctly, even in the presence of a transient fault. The translation essentially triples all basic operations and uses majority voting for operations that may change the control-flow of the program. An essential part of the dynamic semantics of Walker et al.'s faulty lambda calculus is the primitive voting operation, which is used to guarantee correct control-flow behaviour, for function calls and conditionals, in the presence of transient faults. The voting operation is trusted in the sense that Walker et al.'s faulty lambda calculus does not consider the effects of transient faults during voting. This setup suggests that special hardware support may be needed for voting.

The work by Walker et al. on the faulty lambda calculus assumes the Single Event Upset (SEU) fault model, which suggests that at most one fault occurs during evaluation. In this work, we adopt the SEU fault model and extend Walker et al.'s work by giving a translation to a refined lambda calculus that does not provide a built-in voting mechanism, but rather provides two simple operations that are sufficient for compiling away voting in the translation and are *atomic* in the sense that it is assumed that no transient faults corrupt the operations. Intuitively, the first atomic operation allocates a function closure and stores the closure address in three distinct registers. The second atomic operation checks atomically whether two addresses to closures are identical and if so calls the function denoted by the closure. In case the addresses are different, an alternative expression is evaluated.

The two atomic operations make explicit the assumptions of our approach. Whereas the first atomic operation can be implemented atomically on traditional architectures, using a few testing and fault-correcting operations, special architectural support is needed for the second atomic operation for 100 percent safety with respect to the SEU fault model. Without special architectural support, a transient fault on the comparison flag or on the target address during the comparison and branch code sequence can pass non-detected. However, making the assumptions explicit makes it straightforward to reason about the probability of a non-detected (and non-corrected) transient fault.

The contributions of our work is illustrated with the following diagram:

$$\begin{tikzcd}
S \arrow[rr, phantom, "\hookrightarrow"] \arrow[d, Rightarrow] & & S' \arrow[d, Rightarrow] \\
S_0 \arrow[r, phantom, "\rightarrow_0^*"] & S_i \arrow[r, phantom, "\rightarrow_0^*"] \arrow[d, phantom, "\downarrow_1"] & S_n \arrow[d, phantom, "\curlyvee"] \\
 & S^z \arrow[r, phantom, "\rightarrow_0^*"] & S^z_n
\end{tikzcd}$$

First assume that S is a source language machine state that evaluates in one step into a source language machine state S'. Then translating S into the target language equipped with atomic operations for fault-tolerant computing yields S_0 that, in the presence of no faults (indicated with the 0 on the relation $\rightarrow_0^*$), evaluates in zero or more steps into the machine state S_n, which is also the result of translating S'. Now, if at any point during evaluation of S_0 to S_n, say at machine state S_i, a transient fault occurs (indicated with the 1 on the relation $\rightarrow_1$), which will render the system in a machine state S^z (z for zap), then evaluation can proceed to a machine state S^z_n, which essentially contains enough information to extract the

IT University of Copenhagen Technical Report. June 2007.

machine state S'. In particular, if S' is essentially an integer value d then a majority voting applied to S^{z}_n will result in the integer d.

Whereas the faulty translation we propose is similar to that of Walker et al. [18] in that it triples all basic operations, the translation differs in the way function application and conditional expressions (operations that may change control-flow) are translated. Instead of relying on the dynamic semantics of the target language to do the voting, our faulty translation generates code for the voting. Effectively, a conditional expression

$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$

is translated into the following code:

```
let ⟨$x_1, x_2, x_3$⟩ = $e'$
in  if $x_1$ then if $x_2$ then $e'_1$
                  else if $x_3$ then $e'_1$ else $e'_2$
    else if $x_2$ then if $x_3$ then $e'_1$ else $e'_2$
         else $e'_2$
```

Here e', e'_1, and e'_2 are the results of translating e, e_1, and e_2, respectively, and the `let` construct binds the three results of evaluating the faulty translation of e. Notice that the code is safe up-to one fault in any of the values x_1, x_2, and x_3; if at most one fault has occurred, before or during voting, the correct branch is taken.

One artifact of the approach taken here is that the soundness proof relies heavily on the translation and not only on the type system for the target language. Because a fault can occur at any step in the dynamic semantics of the target program, the soundness proof has to consider all possibilities, even faults occurring during voting. It is an open problem how to capture correctness of generated voting code in a type system (i.e., that any single fault will lead to correct results).

In the present setup, the technique assumes fault-safe control registers for holding the instruction pointer, the stack pointer, and the environment for the current closure. In this respect, the setup here is not different from that of [18], from which we have borrowed the particular machine state model.

For self-containedness and clarity, we have chosen to present the technique for incorporating voting in the generated code as a translation from a standard simply typed lambda calculus to an extended simply typed lambda calculus with functions that take multiple arguments and return multiple values and atomic primitives for implementing voting. Another equivalent approach would be to use Walker et al.'s λ_{zap} [18] as the source language for a translation into our extended language and show that the built-in voting primitives in the λ_{zap} language can be eliminated and translated into equivalent voting code in the extended language.

1.1 Outline

In Section 1.2, we first give an overview of related work.

In Section 2, we briefly present the source language of the translation, a simply typed lambda calculus with booleans and integers. We provide a dynamic contextual small-step semantics as well as a type system for the language and state a type soundness result saying that "well-typed programs do not get stuck".

In Section 3, we present an extended lambda calculus, which supports functions that take multiple arguments and return multiple results. The language also supports two new atomic operations that are sufficient for incorporating in-lined voting.

In Section 4, we present a translation from the source language to the extended language and prove that the result of the translation is sound with respect to the semantics of the source language and that a single transient fault during execution of an extended language program results in three values for which two of the values agree with the result of the non-faulty semantics of the source language program.

Finally, in Section 5, we describe future work and conclude.

1.2 Related Work

Most related to this work is the work by Walker et al. on which the present work builds [18]. Whereas Walker et al. focus on deriving a type system for the intermediate language, the focus in this paper is on generating fault-tolerant code for in-lined voting and to establish a soundness result that expresses that in-lined voting code works correctly in the presence of any single transient fault occurring during evaluation.

Also very much related to our work is the work by Perry et al. on a fault-tolerant typed assembly language [8], a type-based framework for ensuring that transient faults in the processor do not go undetected. Whereas the work on a typed assembly language for fault-tolerance seems very promising, their framework does not provide full fault-tolerance in the sense that programs are guaranteed to operate correctly even in the presence of a single fault. For such a guarantee, their framework would be required to incorporate some kind of voting mechanism. A delicate issue in using a typed assembly language for our compilation of voting, would be to support, in the type system, propagation of comparison results in such a way that appropriate (sound) voting code would be enforced by the type system.

Several previous approaches to software-implemented fault detection and fault tolerance have been covered in the literature [9, 10, 12, 11, 13, 5].

In [9], Rebaudengo at al. make use of data duplication and insertion of control-flow check code to harden software by detecting data faults and control-flow faults (faults occurring during execution of conditional statements and jumps between basic blocks). In [10], Rebaudengo at al. extend their previous work by using data duplication and check-sums to recover from transient data faults.

None of the above mentioned approaches, except the approach taken in [18], have been verified formally in terms of correctness or soundness results.

Another line of related work is work on policy based techniques, such as software fault isolation (SFI) [17, 6], control-flow integrity (CFI) [1, 2], and hybrids [16, 15]. None of these techniques, however, protect against transient faults occurring immediately after evaluation of protection code (i.e., check code or sand-boxing code) or against faults occurring within the boundaries of the sand-boxing or during execution of sand-boxing code.

At a technical level, the work here is also related to the work by Yoshida et al. on logical reasoning for higher-order functions with local state [19]. In our work (and in the work by Walker et al. [18]), machine states $(M; e)$, where M maps labels to values and e is an expression, are considered identical up-to consistent renaming of labels. In the work by Yoshida et al., the same equalities are assured by extending machine states to be on the form $\nu\tilde{l}.(M; e)$, where the first part existentially quantifies over the label set $\tilde{l}$ to provide implicit renaming. Whereas we need the property to enforce only one kind of non-determinacy in the dynamic semantics (fault introduction), Yoshida et al. also use the property to ease the reasoning about when two states are observationally equivalent; programs that differ only in the order memory is allocated should result in identical machine states.

2. The Source Language

We assume a denumerable infinite set of labels, ranged over by l, and a denumerable infinite set of variables, ranged over by x. The grammar for source language values and source language expressions is given as follows:

$$\begin{array}{rcl} v & ::= & l \mid \texttt{true*} \mid \texttt{false*} \mid n\texttt{*} \\ e & ::= & x \mid v \mid n \mid \texttt{true} \mid \texttt{false} \\ & \mid & \lambda x.e \mid e_1\, e_2 \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\ & \mid & e_1\ op\ e_2 \quad op \in \{\texttt{+}, \texttt{<}\} \end{array}$$

Notice that the grammar distinguishes between values (e.g., `true*`) and expressions evaluating to a value in one step (e.g., `true`).

2.1 Source Language Dynamic Semantics

For the source language, *evaluation contexts*, ranged over by E, take the following form:

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid \texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ e \\ & \mid & \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ & \mid & E\ op\ e \mid v\ op\ E \end{array}$$

When E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression resulting from filling the hole in E with e.

Memory maps, ranged over by M, map labels to lambda expressions

$$M \quad ::= \quad \varepsilon \mid M, l \mapsto \lambda x.e \qquad l \notin \mathrm{Dom}(M)$$

Machine states, ranged over by S, are pairs $(M; e)$ of a memory map M and an expression e. A *terminal* machine state, S_{v}, is a machine state of the form $(M; v)$, where M is a memory map and v is a value. Single-step reductions take the form $S \hookrightarrow S'$, where S and S' are machine states. We consider machine states identical up-to consistent renaming of labels, which makes allocation in the dynamic semantics deterministic; we shall discuss this issue later in Section 4.5.

Small Step Reductions $\boxed{M; e \hookrightarrow M'; e'}$

$$\frac{M; e \hookrightarrow M'; e' \quad E \neq [\cdot]}{M; E[e] \hookrightarrow M'; E[e']} \tag{1}$$

$$\frac{}{M; \lambda x.e \hookrightarrow (M, l \mapsto \lambda x.e); l} \tag{2}$$

$$\frac{M(l) = \lambda x.e}{M; l\ v \hookrightarrow M; e[v/x]} \tag{3}$$

$$\frac{}{M; \texttt{if true* then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow M; e_1} \tag{4}$$

$$\frac{v \neq \texttt{true*}}{M; \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow M; e_2} \tag{5}$$

$$\frac{}{M; \texttt{false} \hookrightarrow M; \texttt{false*}} \tag{6}$$

$$\frac{}{M; \texttt{true} \hookrightarrow M; \texttt{true*}} \tag{7}$$

$$\frac{}{M; n \hookrightarrow M; n\texttt{*}} \tag{8}$$

$$\frac{}{M; \texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \hookrightarrow M; e[v/x]} \tag{9}$$

$$\frac{n = n_1 + n_2}{M; n_1\texttt{*} \texttt{ + } n_2\texttt{*} \hookrightarrow M; n\texttt{*}} \tag{10}$$

$$\frac{n_1 < n_2}{M; n_1\texttt{*} \texttt{ < } n_2\texttt{*} \hookrightarrow M; \texttt{true*}} \tag{11}$$

$$\frac{n_2 \leq n_1}{M; n_1\texttt{*} \texttt{ < } n_2\texttt{*} \hookrightarrow M; \texttt{false*}} \tag{12}$$

The transitive, reflexive closure of $\hookrightarrow$, written $\hookrightarrow^*$, is defined by the following two rules:

$$\frac{S \hookrightarrow S' \quad S' \hookrightarrow^* S''}{S \hookrightarrow^* S''} \tag{13}$$

$$\frac{}{S \hookrightarrow^* S} \tag{14}$$

We further define $S \uparrow$ to mean that there exists an infinite sequence $S \hookrightarrow S_1 \hookrightarrow S_2 \hookrightarrow \cdots$. Moreover, when $S = (M; e)$ is some machine state and E is an evaluation context, we write $E[S]$ to mean $(M; E[e])$.

2.2 Source Language Type System

Types, ranged over by τ take the following form:

$$\tau \quad ::= \quad \mathit{bool} \mid \mathit{nat} \mid \tau \to \tau'$$

Type assumptions Γ map variables and labels to types:

$$\Gamma \quad ::= \quad \Gamma, x : \tau \mid \Gamma, l : \tau \mid \varepsilon$$

The type system allows inferences among sentences of the form $\Gamma \vdash^{\mathrm{s}} e : \tau$, which are read: "under the assumptions Γ, the expression e has type τ."

Typing Judgments for Expressions $\boxed{\Gamma \vdash^{\mathrm{s}} e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^{\mathrm{s}} x : \tau} \tag{15}$$

$$\frac{\Gamma(l) = \tau}{\Gamma \vdash^{\mathrm{s}} l : \tau} \tag{16}$$

$$\frac{}{\Gamma \vdash^{\mathrm{s}} \texttt{true*} : \mathit{bool}} \tag{17}$$

$$\frac{}{\Gamma \vdash^{\mathrm{s}} \texttt{false*} : \mathit{bool}} \tag{18}$$

$$\frac{}{\Gamma \vdash^{\mathrm{s}} \texttt{true} : \mathit{bool}} \tag{19}$$

$$\frac{}{\Gamma \vdash^{\mathrm{s}} \texttt{false} : \mathit{bool}} \tag{20}$$

$$\frac{}{\Gamma \vdash^{\mathrm{s}} n\texttt{*} : \mathit{nat}} \tag{21}$$

$$\frac{}{\Gamma \vdash^{\mathrm{s}} n : \mathit{nat}} \tag{22}$$

$$\frac{\Gamma \vdash^{\mathrm{s}} e : \mathit{bool} \quad \Gamma \vdash^{\mathrm{s}} e_1 : \tau \quad \Gamma \vdash^{\mathrm{s}} e_2 : \tau}{\Gamma \vdash^{\mathrm{s}} \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau} \tag{23}$$

$$\frac{\Gamma \vdash^{\mathrm{s}} e_1 : \tau \quad \Gamma, (x:\tau) \vdash^{\mathrm{s}} e_2 : \tau'}{\Gamma \vdash^{\mathrm{s}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau'} \tag{24}$$

$$\frac{\Gamma \vdash^{\mathrm{s}} e_1 : \tau \to \tau' \quad \Gamma \vdash^{\mathrm{s}} e_2 : \tau}{\Gamma \vdash^{\mathrm{s}} e_1\ e_2 : \tau'} \tag{25}$$

$$\frac{\Gamma, (x:\tau) \vdash e : \tau'}{\Gamma \vdash^{\mathrm{s}} \lambda x.e : \tau \to \tau'} \tag{26}$$

$$\frac{\Gamma \vdash^{\mathrm{s}} e_1 : \mathit{nat} \quad \Gamma \vdash^{\mathrm{s}} e_2 : \mathit{nat}}{\Gamma \vdash^{\mathrm{s}} e_1 \mathtt{<} e_2 : \mathit{bool}} \tag{27}$$

$$\frac{\Gamma \vdash^{\mathrm{s}} e_1 : \mathit{nat} \quad \Gamma \vdash^{\mathrm{s}} e_2 : \mathit{nat}}{\Gamma \vdash^{\mathrm{s}} e_1 \mathtt{+} e_2 : \mathit{nat}} \tag{28}$$

Following the approach of [18], typing of machine states is defined by point-wise typing of memory maps.

Typing Judgments for Memory $\boxed{\vdash^{\mathrm{s}} M : \Gamma}$

$$\frac{\begin{array}{c}\mathrm{Dom}(M) = \mathrm{Dom}(\Gamma) \\ \text{for all } l \in \mathrm{Dom}(M). \\ M(l) = \lambda x.e \\ \Gamma(l) = \tau \to \tau' \\ \Gamma, (x:\tau) \vdash^{\mathrm{s}} e : \tau'\end{array}}{\vdash^{\mathrm{s}} M : \Gamma} \tag{29}$$

The expression part e of a machine state $(M; e)$ is typed under the type assumptions provided by the memory map M.

Typing Judgments for Machine States $\boxed{\vdash^{\mathrm{s}} (M; e) : \tau}$

$$\frac{\vdash^{\mathrm{s}} M : \Gamma \quad \Gamma \vdash^{\mathrm{s}} e : \tau}{\vdash^{\mathrm{s}} (M; e) : \tau} \tag{30}$$

2.3 Properties of the Source Language

We first give a few definitions before we present a unique decomposition proposition, which is used for the proofs of type preservation and progress [7].

A *redex*, ranged over by r, is an expression of the form

$$\begin{array}{rcl} r & ::= & n \mid l\ v \mid \mathtt{let}\ x = v\ \mathtt{in}\ e \mid \lambda x.e \\ & \mid & \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{true} \mid \mathtt{false} \\ & \mid & v_1\ \mathit{op}\ v_2 \quad \mathit{op} \in \{\mathtt{+}, \mathtt{<}\} \end{array}$$

The following unique decomposition proposition states that any well-typed term is either a value or the term can be decomposed into a unique context and a unique well-typed redex. Moreover, filling the context with an expression of the same type as the redex results in a well-typed term:

PROPOSITION 1 (Unique Decomposition). *If* $\vdash^{\mathrm{s}} (M; e) : \tau$ *then either* e *is a value* v *or there exists a unique* E*, a unique redex* e'*, and a unique* τ' *such that* $e = E[e']$ *and* $\vdash^{\mathrm{s}} (M; e') : \tau'$*. Furthermore, for all* e'' *such that* $\vdash^{\mathrm{s}} (M; e'') : \tau'$*, it follows that* $\vdash^{\mathrm{s}} (M; E[e'']) : \tau$*.*

PROOF By induction over the derivation $\Gamma \vdash^{\mathrm{s}} e : \tau$, where $\vdash^{\mathrm{s}} M : \Gamma$. □

The proofs of the following type preservation and progress propositions is then straightforward and standard [7].

PROPOSITION 2 (Type Preservation). *If* $\vdash^{\mathrm{s}} S : \tau$ *and* $S \hookrightarrow S'$ *then* $\vdash^{\mathrm{s}} S' : \tau$*.*

PROOF By induction over the structure of the expression-part of S, using Proposition 1. □

PROPOSITION 3 (Progress). *If* $\vdash^{\mathrm{s}} S : \tau$ *then either (1)* S *is a terminal machine state* S_{v}*; or (2) there exists* S' *such that* $S \hookrightarrow S'$*.*

PROOF By induction over the structure of the expression-part of S, using Proposition 1. □

PROPOSITION 4 (Type Soundness). *If* $\vdash^{\mathrm{s}} S : \tau$ *then either* $S\!\uparrow$ *or there exists a terminal machine state* S_{v} *such that* $S \hookrightarrow^* S_{\mathrm{v}}$*.*

PROOF By induction on the number of machine steps using Proposition 2 and Proposition 3. □

3. The Extended Language

We now extend the source language to allow functions to take multiple arguments and return multiple results. We use c to range over a finite set of colors. We use vs to denote (colored) *value sequences* $\langle v_1 \triangleright c_1, \cdots, v_n \triangleright c_n\rangle$, $n \geq 0$ and xs to denote (colored) *variable sequences* $\langle x_1 \triangleright c_1, \cdots, x_n \triangleright c_n\rangle$, $n \geq 0$. Moreover, we use cs to denote *color sequences* $\langle c_1, \cdots, c_n\rangle$, $n \geq 0$. Intuitively, colors are used in what follows to distinguish different copies of tripled computation.

We also extend the source language with two atomic operations that capture precisely the assumptions of the approach, as described in the introduction.

Intuitively, the first atomic operation $\lambda^{\mathit{cs}}\mathit{xs}.e$ simultaneously loads the same function label into a sequence of distinct colored registers, as specified by the color sequence cs. The source language lambda expression $\lambda x.e$ can be encoded in the extended language by the expression $\lambda^{\mathit{cs}}\langle x\rangle.e'$, where e' is the extended-language version of e and where cs is a singleton color sequence.

The second atomic operation $\mathtt{appe}\ [l_1, l_2]\ a\ \mathtt{else}\ e$ atomically checks if l_1 and l_2 are identical labels and if so calls the function denoted by l_1 and l_2 with a as argument. In case the labels are different, the expression e is evaluated instead.

Notice that atomicity here refers to the requirement that, for 100 percent soundness of the approach, the operation must operate faithfully even if a transient fault occurs during the operation.

The grammar for the extended language is as follows:

$$\begin{array}{rcl} v & ::= & l \mid \mathtt{true*} \mid \mathtt{false*} \mid n\mathtt{*} \\ w & ::= & v \triangleright c \\ \mathit{vs} & ::= & w \mid \langle w_1, \cdots, w_n\rangle \quad n \geq 0 \\ \mathit{cs} & ::= & \langle c_1, \cdots, c_n\rangle \quad n \geq 0 \\ \mathit{xs} & ::= & \langle x_1 \triangleright c_1, \cdots, x_n \triangleright c_n\rangle \quad n \geq 0 \\ \mathit{op} & ::= & \mathtt{+} \mid \mathtt{<} \\ e & ::= & x \triangleright c \mid w \mid n \triangleright c \mid \mathtt{true} \triangleright c \mid \mathtt{false} \triangleright c \\ & \mid & \lambda^{\mathit{cs}}\mathit{xs}.e \mid \mathtt{appe}\ [e_1, e_2]\ e\ \mathtt{else}\ e' \\ & \mid & e_1\ e_2 \mid e\ \mathit{op}_c\ e' \mid \langle e_1, \cdots, e_n\rangle \quad n \geq 0 \\ & \mid & \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{let}\ \mathit{xs} = e_1\ \mathtt{in}\ e_2 \end{array}$$

Assuming $\mathit{xs} = \langle x_1 \triangleright c_1, \cdots, x_n \triangleright c_n\rangle$ and $\mathit{vs} = \langle v_1 \triangleright c_1, \cdots, v_n \triangleright c_n\rangle$ and provided e is an extended language expression, we define the *substitution of vs for xs in* e, written $e[\mathit{vs}/\mathit{xs}]$, as the simultaneous substitution of pairs $v_i \triangleright c_i$ for free occurrences of pairs $x_i \triangleright c_i$ in e, assuming $1 \leq i \leq n$. Notice that substitutions, according to the above definition, must be color preserving.

3.1 Dynamic Semantics

Evaluation contexts, ranged over by E, take the following form:

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid E\,e \mid w\,E \mid \texttt{appe}\,[E,e]\,e'\,\texttt{else}\,e'' \\
 & \mid & \texttt{appe}\,[w,E]\,e\,\texttt{else}\,e' \mid \texttt{appe}\,[w,w']\,E\,\texttt{else}\,e \\
 & \mid & \texttt{let}\,\mathit{xs} = E\,\texttt{in}\,e \mid \texttt{if}\,E\,\texttt{then}\,e_1\,\texttt{else}\,e_2 \\
 & \mid & E\;op_c\;e \mid w\;op_c\;E \\
 & \mid & \langle w_1, ..., w_n, E, e_1, ..., e_m\rangle \quad n, m \geq 0
\end{array}
$$

As for the source language, when E is an evaluation context and e is an expression, we write $E[e]$ to denote the expression resulting from filling the hole in E with e.

Memory maps, ranged over by M, map labels to lambda expressions

$$M \quad ::= \quad \varepsilon \mid M, l \mapsto \lambda \mathit{xs}.e \qquad l \notin \mathrm{Dom}(M)$$

As for the source language, *machine states*, ranged over by S, are pairs $(M; e)$ of a memory map M and an expression e. A *terminal* (extended language) machine state S_v is a machine state on the form $(M; \mathit{vs})$, for some memory map M and value sequence vs. Single-step, possibly faulty, reductions take the form $S \longrightarrow_k S'$, where k is either 0 or 1 (indicating whether a transient fault has occurred) and where S and S' are machine states. As for the source language, extended language machine states are considered identical up-to consistent renaming of labels. Treating machine states identical up-to consistent renaming of labels is essential for establishing appropriate fault tolerance properties; we shall come back to this issue in Section 4.5.

Small Step Reductions $\boxed{M; e \longrightarrow_k M'; e'}$

$$\frac{n = n_1 + n_2}{M; n_1\texttt{*} \rhd c_1 \texttt{+}_c\, n_2\texttt{*} \rhd c_2 \longrightarrow_0 M; n\texttt{*} \rhd c} \quad (31)$$

$$\frac{w_i \neq n \rhd c',\ i \in \{1, 2\}}{M; w_1 \texttt{+}_c\, w_2 \longrightarrow_0 M; v \rhd c} \quad (32)$$

$$\frac{n_1 < n_2}{M; n_1\texttt{*} \rhd c_1 \texttt{<}_c\, n_2\texttt{*} \rhd c_2 \longrightarrow_0 M; \texttt{true*} \rhd c} \quad (33)$$

$$\frac{n_2 \leq n_1}{M; n_1\texttt{*} \rhd c_1 \texttt{<}_c\, n_2\texttt{*} \rhd c_2 \longrightarrow_0 M; \texttt{false*} \rhd c} \quad (34)$$

$$\frac{w_i \neq n \rhd c',\ i \in \{1, 2\}}{M; w_1 \texttt{<}_c\, w_2 \longrightarrow_0 M; v \rhd c} \quad (35)$$

$$\frac{\mathit{vs} = \langle l_1 \rhd c_1, \cdots, l_n \rhd c_n\rangle \quad l_i = l}{M; \lambda^{\langle c_1, \cdots, c_n\rangle} \mathit{xs}.e \longrightarrow_0 (M, l \mapsto \lambda \mathit{xs}.e); \mathit{vs}} \quad (36)$$

$$\frac{M(l) = \lambda \mathit{xs}.e}{M; (l \rhd c)\ \mathit{vs} \longrightarrow_0 M; e[\mathit{vs}/\mathit{xs}]} \quad (37)$$

$$\frac{}{M; \texttt{if}\ \texttt{true*} \rhd c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow_0 M; e_1} \quad (38)$$

$$\frac{w \neq \texttt{true*} \rhd c}{M; \texttt{if}\ w\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow_0 M; e_2} \quad (39)$$

$$\frac{}{M; \texttt{false} \rhd c \longrightarrow_0 M; \texttt{false*} \rhd c} \quad (40)$$

$$\frac{}{M; \texttt{true} \rhd c \longrightarrow_0 M; \texttt{true*} \rhd c} \quad (41)$$

$$\frac{}{M; n \rhd c \longrightarrow_0 M; n\texttt{*} \rhd c} \quad (42)$$

$$\frac{}{M; \texttt{let}\ \mathit{xs} = \mathit{vs}\ \texttt{in}\ e \longrightarrow_0 M; e[\mathit{vs}/\mathit{xs}]} \quad (43)$$

$$\frac{M(l) = \lambda \mathit{xs}.e}{M; \texttt{appe}\ [l \rhd c, l \rhd c']\ \mathit{vs}\ \texttt{else}\ e' \longrightarrow_0 M; e[\mathit{vs}/\mathit{xs}]} \quad (44)$$

$$\frac{v_1 \neq v_2}{M; \texttt{appe}\ [v_1 \rhd c, v_2 \rhd c']\ \mathit{vs}\ \texttt{else}\ e \longrightarrow_0 M; e} \quad (45)$$

$$\frac{M; e \longrightarrow_k M'; e' \quad E \neq [\cdot]}{M; E[e] \longrightarrow_k M'; E[e']} \quad (46)$$

$$\frac{}{M; E[v \rhd c] \longrightarrow_1 M; E[v' \rhd c]} \quad (47)$$

Notice that (47) implements a Single Event Upset (SEU) model [18]; the rule allows for a single transient fault to alter a value on the stack or a value in a register.

Notice also that because substitutions must preserve colors, to apply (37), (43), and (44), it must first be established that *vs* and *xs* agree on colors.

Further notice that colors have no influence on the reductions and are only there for proving properties about reductions. An implementation may therefore erase colors from terms before execution.

The transitive, reflexive closure of $\longrightarrow_k$, written $\longrightarrow_k^*$, is defined by the following two rules:

$$\frac{S \longrightarrow_j S' \quad S' \longrightarrow_k^* S''}{S \longrightarrow_{j+k}^* S''} \quad (48)$$

$$\frac{}{S \longrightarrow_0^* S} \quad (49)$$

We further define $S \uparrow_k$ to mean that there exists an infinite sequence $S \longrightarrow_k^* S_1 \hookrightarrow_0 S_2 \hookrightarrow_0 \cdots$. As for the source language, when $S = (M; e)$ is some machine state and E is an evaluation context, we write $E[S]$ to mean $(M; E[e])$.

3.2 A Type System for the Faulty Language

We now present a simple type system for the extended language. The type system associates terms with colors and is similar to the type system for the faulty lambda calculus, developed in [18].

Types, ranged over by τ, and *type and color sequences* $\langle \tau_1 \rhd c_1, \cdots, \tau_n \rhd c_n\rangle$, ranged over by κ, take the following form:

$$
\begin{array}{rcl}
\kappa & ::= & \langle \tau_1 \rhd c_1, \cdots, \tau_n \rhd c_n\rangle \mid \tau \rhd c \\
\tau & ::= & \mathit{bool} \mid \mathit{nat} \mid \kappa \rightarrow \kappa'
\end{array}
$$

Type assumptions Γ map labels to types and variables to types and colors:

$$\Gamma \quad ::= \quad \Gamma, l : \tau \mid \Gamma, x : \tau \rhd c \mid \varepsilon$$

The type system allows inferences among sentences of the form $\Gamma \vdash e : \kappa$, which are read: "under the assumptions Γ, the expression e has type and color sequence κ."

We often write $(\mathit{xs} : \kappa)$ to denote the type assumptions $(x_1 : \tau_1 \rhd c_1, \cdots, x_n : \tau_n \rhd c_n)$ when $\mathit{xs} = \langle x_1 \rhd c_1, \cdots, x_n \rhd c_n\rangle$ and $\kappa = \langle \tau_1 \rhd c_1, \cdots, \tau_n \rhd c_n\rangle$ for some colors $c_1, \cdots, c_n$.

Typing Judgments for Expressions $\boxed{\Gamma \vdash e : \kappa}$

$$\frac{\Gamma(x) = \tau \triangleright c}{\Gamma \vdash x \triangleright c : \tau \triangleright c} \tag{50}$$

$$\frac{\Gamma(l) = \tau}{\Gamma \vdash l \triangleright c : \tau \triangleright c} \tag{51}$$

$$\frac{}{\Gamma \vdash \texttt{true*} \triangleright c : \mathit{bool} \triangleright c} \tag{52}$$

$$\frac{}{\Gamma \vdash \texttt{false*} \triangleright c : \mathit{bool} \triangleright c} \tag{53}$$

$$\frac{}{\Gamma \vdash \texttt{true} \triangleright c : \mathit{bool} \triangleright c} \tag{54}$$

$$\frac{}{\Gamma \vdash \texttt{false} \triangleright c : \mathit{bool} \triangleright c} \tag{55}$$

$$\frac{}{\Gamma \vdash n\texttt{*} \triangleright c : \mathit{nat} \triangleright c} \tag{56}$$

$$\frac{}{\Gamma \vdash n \triangleright c : \mathit{nat} \triangleright c} \tag{57}$$

$$\frac{\Gamma \vdash e : \mathit{bool} \quad \Gamma \vdash e_1 : \kappa \quad \Gamma \vdash e_2 : \kappa}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \kappa} \tag{58}$$

$$\frac{\Gamma \vdash e_1 : \kappa \quad \Gamma, (\mathit{xs} : \kappa) \vdash e_2 : \kappa'}{\Gamma \vdash \texttt{let}\ \mathit{xs} \texttt{=} e_1\ \texttt{in}\ e_2 : \kappa'} \tag{59}$$

$$\frac{\Gamma \vdash e_1 : \kappa \to \kappa' \triangleright c \quad \Gamma \vdash e_2 : \kappa}{\Gamma \vdash e_1\ e_2 : \kappa'} \tag{60}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \kappa \to \kappa' \triangleright c_1 \quad \Gamma \vdash e_2 : \kappa \to \kappa' \triangleright c_2 \\ \Gamma \vdash e : \kappa \quad \Gamma \vdash e' : \kappa'\end{array}}{\Gamma \vdash \texttt{appe}\ [e_1, e_2]\ e\ \texttt{else}\ e' : \kappa'} \tag{61}$$

$$\frac{\begin{array}{c}\Gamma, (\mathit{xs} : \kappa) \vdash e : \kappa' \\ \tau_i = \kappa \to \kappa' \quad i = [1..n]\end{array}}{\Gamma \vdash \lambda^{\langle c_1, \cdots, c_n \rangle} \mathit{xs}.e : \langle \tau_1 \triangleright c_1, \cdots, \tau_n \triangleright c_n \rangle} \tag{62}$$

$$\frac{\Gamma \vdash e_i : \tau_i \triangleright c_i \quad i = [1..n]}{\Gamma \vdash \langle e_1, \cdots, e_n \rangle : \langle \tau_1 \triangleright c_1, \cdots, \tau_n \triangleright c_n \rangle} \tag{63}$$

$$\frac{\Gamma \vdash e_i : \mathit{nat} \triangleright c_i \quad i = [1, 2]}{\Gamma \vdash e_1 \texttt{+}_c\ e_2 : \mathit{nat} \triangleright c} \tag{64}$$

$$\frac{\Gamma \vdash e_i : \mathit{nat} \triangleright c_i \quad i = [1, 2]}{\Gamma \vdash e_1 \texttt{<}_c\ e_2 : \mathit{bool} \triangleright c} \tag{65}$$

Typing Judgments for Memory $\boxed{\vdash M : \Gamma}$

$$\frac{\begin{array}{c}\mathrm{Dom}(M) = \mathrm{Dom}(\Gamma) \\ \text{for all } l \in \mathrm{Dom}(M). \\ M(l) = \lambda \mathit{xs}.e \\ \Gamma(l) = \kappa \to \kappa' \\ \Gamma, (\mathit{xs} : \kappa) \vdash e : \kappa'\end{array}}{\vdash M : \Gamma} \tag{66}$$

Typing Judgments for Machine States $\boxed{\vdash (M; e) : \kappa}$

$$\frac{\vdash M : \Gamma \quad \Gamma \vdash e : \kappa}{\vdash (M; e) : \kappa} \tag{67}$$

3.3 Properties of the Extended Language

We now carry over the results of Section 2.3 to the extended language.

A *redex*, ranged over by r, is an expression of the form

$$\begin{array}{rcl} r & ::= & n \triangleright c \mid w\ \mathit{vs} \mid \texttt{let}\ \mathit{xs} \texttt{=} \mathit{vs}\ \texttt{in}\ e \mid \lambda^{\mathit{cs}} \mathit{xs}.e \\ & \mid & \texttt{if}\ v \triangleright c\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{true} \triangleright c \mid \texttt{false} \triangleright c \\ & \mid & w_1\ \mathit{op}_c\ w_2 \quad \mathit{op} \in \{\texttt{+}, \texttt{<}\} \\ & \mid & \texttt{appe}\ [w_1, w_2]\ \mathit{vs}\ \texttt{else}\ e \end{array}$$

PROPOSITION 5 (Unique Decomposition). *If* $\vdash (M; e) : \kappa$ *then either* e *is a value sequence* vs *or there exists a unique* E*, a unique redex* e'*, and a unique* τ' *such that* $e = E[e']$ *and* $\vdash (M; e') : \tau'$*. Furthermore, for all* e'' *such that* $\vdash (M; e'') : \tau'$*, it follows that* $\vdash (M; E[e'']) : \kappa$*.*

PROOF By induction over the derivation $\Gamma \vdash e : \kappa$, where $\vdash M : \Gamma$. □

PROPOSITION 6 (Type Preservation). *If* $\vdash S : \kappa$ *and* $S \to_0 S'$ *then* $\vdash S' : \kappa$

PROOF By induction over the structure of the expression-part of S, using Proposition 5. □

PROPOSITION 7 (Progress). *If* $\vdash S : \kappa$ *then either (1)* S *is a terminal machine state* S_{v}*; or (2) there exists* S' *such that* $S \to_0 S'$*.*

PROOF By induction over the structure of the expression-part of S, using Proposition 5. □

PROPOSITION 8 (Type Soundness). *If* $\vdash S : \tau$ *then either* $S{\uparrow}_0$ *or there exists a terminal machine state* S_{v} *such that* $S \to_0^* S_{\mathrm{v}}$*.*

PROOF By induction on the number of machine steps using Proposition 6 and Proposition 7. □

The following proposition expresses that evaluation is deterministic, which depends crucially on the fact that machine states are considered identical up-to consistent renaming of labels.

PROPOSITION 9 (Evaluation is Deterministic). *If* $S \to_0 S'$ *and* $S \to_0 S''$ *then* $S = S''$*.*

PROOF By induction on the structure of e in S. □

This proposition is used later in the proof of the fault-tolerance property.

4. The Faulty Translation

A *translation environment* (ρ) is a mapping from identifiers to triples of identifiers:

$$\begin{array}{rcll} \rho & ::= & \rho', x \mapsto (x_1, x_2, x_3) & x \notin \mathrm{Dom}(\rho') \wedge \\ & & & x_i = x_j \Rightarrow i = j \wedge \\ & & & x_i \notin \mathrm{Ran}(\rho') \\ & \mid & \varepsilon & \end{array}$$

The side conditions on valid translation environments are enforced to avoid variable capture.

The faulty translation is defined inductively over source language expressions as a function $[\![e]\!]\rho = e'$, where e is a source

$$\begin{array}{l}\llbracket x \rrbracket\rho = \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle \\ \quad \textit{where } \rho(x) = (x_1, x_2, x_3)\end{array}$$

$$\begin{array}{l}\llbracket \lambda x.e \rrbracket\rho = \lambda^{\langle \mathrm{r},\mathrm{g},\mathrm{b}\rangle} \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle.e' \\ \quad \textit{where } e' = \llbracket e \rrbracket(\rho, x \mapsto (x_1, x_2, x_3))\end{array}$$

$$\llbracket n \rrbracket\rho = \langle\!\langle n, n, n \rangle\!\rangle$$

$$\llbracket \texttt{true} \rrbracket\rho = \langle\!\langle \texttt{true}, \texttt{true}, \texttt{true} \rangle\!\rangle$$

$$\llbracket \texttt{false} \rrbracket\rho = \langle\!\langle \texttt{false}, \texttt{false}, \texttt{false} \rangle\!\rangle$$

$$\begin{array}{l}\llbracket \texttt{let}\ x \texttt{ = } e_1 \ \texttt{in}\ e_2 \rrbracket\rho = \\ \quad \texttt{let}\ \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle \texttt{ = } \llbracket e_1 \rrbracket\rho \\ \quad \texttt{in}\ \llbracket e_2 \rrbracket(\rho, x \mapsto (x_1, x_2, x_3))\end{array}$$

$$\begin{array}{l}\llbracket \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rrbracket\rho = \\ \quad \texttt{let}\ \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle \texttt{ = } \llbracket e \rrbracket\rho \\ \quad \texttt{in}\ \ \texttt{if}\ x_1 \triangleright \mathrm{r}\ \texttt{then} \\ \qquad\qquad \texttt{if}\ x_2 \triangleright \mathrm{g}\ \texttt{then}\ \llbracket e_1 \rrbracket\rho \\ \qquad\qquad \texttt{else if}\ x_3 \triangleright \mathrm{b}\ \texttt{then}\ \llbracket e_1 \rrbracket\rho\ \texttt{else}\ \llbracket e_2 \rrbracket\rho \\ \qquad\quad \texttt{else} \\ \qquad\qquad \texttt{if}\ x_2 \triangleright \mathrm{g}\ \texttt{then}\ \texttt{if}\ x_3 \triangleright \mathrm{b}\ \texttt{then}\ \llbracket e_1 \rrbracket\rho\ \texttt{else}\ \llbracket e_2 \rrbracket\rho \\ \qquad\qquad \texttt{else}\ \llbracket e_2 \rrbracket\rho\end{array}$$

$$\begin{array}{l}\llbracket e_1\ e_2 \rrbracket\rho = \\ \quad \texttt{let}\ \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle \texttt{ = } \llbracket e_1 \rrbracket\rho \\ \quad \texttt{in}\ \ \texttt{appe}\ [x_1 \triangleright \mathrm{r}, x_2 \triangleright \mathrm{g}]\ (\llbracket e_2 \rrbracket\rho) \\ \qquad\quad \texttt{else}\ \ \texttt{appe}\ [x_1 \triangleright \mathrm{r}, x_3 \triangleright \mathrm{b}]\ (\llbracket e_2 \rrbracket\rho) \\ \qquad\qquad\quad \texttt{else}\ (x_2 \triangleright \mathrm{g})\ (\llbracket e_2 \rrbracket\rho)\end{array}$$

$$\begin{array}{l}\llbracket e_1\ \mathit{op}\ e_2 \rrbracket\rho = \qquad\qquad\qquad\qquad\qquad \mathit{op} \in \{\texttt{+}, \texttt{<}\} \\ \quad \texttt{let}\ \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle \texttt{ = } \llbracket e_1 \rrbracket\rho \\ \quad \texttt{in}\ \ \texttt{let}\ \langle\!\langle y_1, y_2, y_3 \rangle\!\rangle \texttt{ = } \llbracket e_2 \rrbracket\rho \\ \qquad \texttt{in}\ \langle x_1 \triangleright \mathrm{r}\ \mathit{op}_{\mathrm{r}}\ y_1 \triangleright \mathrm{r},\ x_2 \triangleright \mathrm{g}\ \mathit{op}_{\mathrm{g}}\ y_2 \triangleright \mathrm{g},\ x_3 \triangleright \mathrm{b}\ \mathit{op}_{\mathrm{b}}\ y_3 \triangleright \mathrm{b} \rangle\end{array}$$

$$\llbracket v \rrbracket\rho = \langle\!\langle v, v, v \rangle\!\rangle$$

Figure 1. Translation of Expressions and Values.

language expression, ρ is a translation environment, and where e' is the result of translating e. For the translation, we assume three colors r, g, and b. We often write $\langle\!\langle A, B, C \rangle\!\rangle$ to mean $\langle A \triangleright \mathrm{r}, B \triangleright \mathrm{g}, C \triangleright \mathrm{b} \rangle$. The translation rules are presented in Figure 1.

Essentially, each primitive operation in the source program is translated into three operations in the target language, corresponding to the three different colors r, g, and b. Variable bindings are also tripled, which means that a single let-binding is translated into a let-binding construct in the target language, binding three distinct variables with different colors r, g, and b. Moreover, arguments to functions are tripled and so are return values from functions. Control-flow operations (i.e., function calls and conditional expressions) translate into code that involves voting. For conditional expressions, the translated code ensures that a branch is taken only if two of the condition computations agree on the branch. For function calls, the translated code ensures that, provided two of the three expressions denoting the function closure agree on a label, then the function denoted by this label is called.

The potential quadratic increase in code size, caused by the duplication of branch expressions, may be eliminated by replacing large branches with calls to thunkified code prior to the translation. In the case for function application, it is straightforward to avoid the duplication caused by repeated translation of the argument expression by introducing a let-binding construct for the argument.

$$\llbracket \mathit{bool} \rrbracket_0 = \mathit{bool}$$

$$\llbracket \mathit{nat} \rrbracket_0 = \mathit{nat}$$

$$\llbracket \tau_1 \to \tau_2 \rrbracket_0 = \llbracket \tau_1 \rrbracket \to \llbracket \tau_2 \rrbracket$$

$$\llbracket \tau \rrbracket = \langle\!\langle \llbracket \tau \rrbracket_0, \llbracket \tau \rrbracket_0, \llbracket \tau \rrbracket_0 \rangle\!\rangle$$

$$\begin{array}{l}\llbracket \Gamma \rrbracket\rho = \Gamma' \\ \quad \textit{where } \mathrm{Dom}(\Gamma) = \mathrm{Dom}(\rho)\ \textit{ and} \\ \qquad\qquad \mathrm{Dom}(\Gamma') = \mathit{fv}(\mathrm{Ran}(\rho))\ \textit{ and} \\ \qquad\qquad \Big((\Gamma(x) = \tau\ \textit{ and }\ \rho(x) = (x_1, x_2, x_3))\ \textit{ implies} \\ \qquad\qquad\quad \llbracket \tau \rrbracket = \langle \Gamma'(x_1), \Gamma'(x_2), \Gamma'(x_3) \rangle\Big)\end{array}$$

Figure 2. Translation of Types and Type Assumptions.

$$\begin{array}{l}\llbracket M \rrbracket = M' \\ \quad \textit{where } \mathrm{Dom}(M) = \mathrm{Dom}(M')\ \textit{ and} \\ \qquad\qquad \Big(M(l) = \lambda x.e\ \textit{ implies} \\ \qquad\qquad\quad M'(l) = \lambda\langle\!\langle x_1, x_2, x_3 \rangle\!\rangle.\llbracket e \rrbracket(x \mapsto (x_1, x_2, x_3))\Big)\end{array}$$

$$\llbracket (M; e) \rrbracket\rho = (\llbracket M \rrbracket; \llbracket e \rrbracket\rho)$$

Figure 3. Translation of Memories and Machine States.

4.1 Properties of the translation

For proving that the translation preserves typings (in a precise sense), we extend the translation to types as shown in Figure 2.

In what follows, extended language substitutions $e[\mathit{vs}/\mathit{xs}]$ are always on a form where $\mathit{vs} = \langle\!\langle v_1, v_2, v_3 \rangle\!\rangle$ and $\mathit{xs} = \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle$, for some values v_1, v_2, and v_3, and some variables x_1, x_2, and x_3. Substitutions of this form are always defined and we shall use this property implicitly in the proofs.

We are now in the position to state a property saying that translation preserves typings:

PROPOSITION 10 (Translation Preserves Typing).

1. If $\Gamma \vdash^{\mathrm{s}} e : \tau$ and $\Gamma' = \llbracket \Gamma \rrbracket\rho$ then $\Gamma' \vdash \llbracket e \rrbracket\rho : \llbracket \tau \rrbracket$.
2. If $\vdash^{\mathrm{s}} S : \tau$ then $\vdash \llbracket S \rrbracket\varepsilon : \llbracket \tau \rrbracket$.

PROOF Part 1 can be proved using induction over the structure of e. Part 2 can be proved directly by unfolding the definitions of machine state typings and machine state translation, using part 1. □

Notice that Proposition 10 also expresses that well-typed source language programs may be translated.

The translation is extended to memories M and machine states $(M; e)$, as shown in Figure 3.

PROPOSITION 11 *If $S \to_0^* S'$ then $E[S] \to_0^* E[S']$.*

PROOF Follows immediately by induction from (46), (48) and (49). □

PROPOSITION 12 (Translation Closed Under Substitution). *If $e' = \llbracket e \rrbracket(\rho, x \mapsto (x_1, x_2, x_3))$ then $e'[\langle\!\langle v, v, v \rangle\!\rangle / \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle] = \llbracket e[v/x] \rrbracket\rho$.*

PROOF By induction over the structure of e. □

PROPOSITION 13 (Translation Substitution Invariance). *If* $e' = [\![e]\!]\rho$ *and* $\{x_1, x_2, x_3\} \cap fv(\mathrm{Ran}(\rho)) = \emptyset$*, it holds for any* v_1, v_2*, and* v_3 *that* $e'[\langle\!\langle v_1, v_2, v_3\rangle\!\rangle/\langle\!\langle x_1, x_2, x_3\rangle\!\rangle] = e'$.

PROOF By induction over the structure of e. $\square$

We can now state a simulation proposition expressing that the translation of a source language program evaluates in alignment with the source language program. This property corresponds to the upper part of the diagram in Section 1, where $\Rightarrow$ corresponds to translation in the empty environment:

$$\begin{tikzcd} S \arrow[rr, hook] \arrow[d, Rightarrow] & & S' \arrow[d, Rightarrow] \\ S_0 \arrow[r, "*"', "\to_0"] & S_i \arrow[r, "\to_0^*"] & S_n \end{tikzcd}$$

The proposition is expressed as follows with $S_0 = [\![S]\!]\varepsilon$ and $S_n = [\![S']\!]\varepsilon$:

PROPOSITION 14 (Simulation). *If* $S \hookrightarrow S'$ *and* $[\![S]\!]\varepsilon$ *and* $[\![S']\!]\varepsilon$ *exist then* $[\![S]\!]\varepsilon \to_0^* [\![S']\!]\varepsilon$.

PROOF By induction over the structure of e, where $S = (M; e)$ and $S' = (M'; e')$.

$\boxed{\textsc{Case}\ e = \mathtt{true}}$ It follows from (7) that $e' = \mathtt{true*}$. The result follows easily from the definition of translation, using (46) and (41) repeatedly.

$\boxed{\textsc{Case}\ e = \mathtt{false}}$ As the case for $e = \mathtt{true}$.

$\boxed{\textsc{Case}\ e = n}$ As the case for $e = \mathtt{true}$.

$\boxed{\textsc{Case}\ e = \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2}$ There are two subcases. Either (9) is applied, in which case e_1 is a value v, or (1) is applied.

In the first subcase, we have $M' = M$ and $e' = e_2[v/x]$. We also have $[\![e]\!]\varepsilon = \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = \langle\!\langle v, v, v\rangle\!\rangle\ \mathtt{in}\ [\![e_2]\!](x \mapsto (x_1, x_2, x_3))$. From (43), it follows that $[\![e]\!]\varepsilon \to_0 [\![e_2]\!](x \mapsto (x_1, x_2, x_3))[\langle\!\langle v, v, v\rangle\!\rangle/\langle\!\langle x_1, x_2, x_3\rangle\!\rangle]$. Further, from Proposition 12, we have $[\![e]\!]\varepsilon \to_0 [\![e_2[v/x]]\!]\varepsilon$ and thus $[\![e]\!]\varepsilon \to_0 [\![e']\!]\varepsilon$. Finally, because $M = M'$, from assumptions and from (49), we have $[\![(M, e)]\!]\varepsilon \to_0^* [\![(M'; e')]\!]\varepsilon$, as required.

In the second subcase, we have from (1) that there exists e_1' such that $(M, e_1) \hookrightarrow (M', e_1')$ and $e' = \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2$. By induction, we have $[\![(M, e_1)]\!]\varepsilon \to_0^* [\![(M'; e_1')]\!]\varepsilon$. We also have $[\![e]\!]\varepsilon = \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![e_1]\!]\varepsilon\ \mathtt{in}\ [\![e_2]\!](x \mapsto (x_1, x_2, x_3))$. Because $[\![e']\!]\varepsilon = \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![e_1']\!]\varepsilon\ \mathtt{in}\ [\![e_2]\!](x \mapsto (x_1, x_2, x_3))$, we can apply Proposition 11 to get $[\![(M, e)]\!]\varepsilon \to_0^* [\![(M'; e')]\!]\varepsilon$, as required.

$\boxed{\textsc{Case}\ e = \mathtt{if}\ e_{\mathrm{b}}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2}$ There are three subcases to consider. Either $e_{\mathrm{b}} = \mathtt{true*}$, or e_{b} is a value different from $\mathtt{true*}$ or e_{b} is not a value.

In the first subcase, where $e_{\mathrm{b}} = \mathtt{true*}$, rule (4) must have been applied, thus, we have $e' = e_1$ and $M' = M$. From the definition of translation, it follows that

$$\begin{array}{rl} [\![e]\!]\varepsilon = & \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = \langle\!\langle \mathtt{true*}, \mathtt{true*}, \mathtt{true*}\rangle\!\rangle \\ & \quad \mathtt{in}\ \mathtt{if}\ x_1 \triangleright \mathrm{r}\ \mathtt{then} \\ & \qquad\qquad \mathtt{if}\ x_2 \triangleright \mathrm{g}\ \mathtt{then}\ [\![e_1]\!]\varepsilon \\ & \qquad\qquad \mathtt{else\ if}\ x_3 \triangleright \mathrm{b}\ \mathtt{then}\ [\![e_1]\!]\varepsilon\ \mathtt{else}\ [\![e_2]\!]\varepsilon \\ & \qquad\quad \mathtt{else} \\ & \qquad\qquad \mathtt{if}\ x_2 \triangleright \mathrm{g}\ \mathtt{then\ if}\ x_3 \triangleright \mathrm{b}\ \mathtt{then}\ [\![e_1]\!]\varepsilon\ \mathtt{else}\ [\![e_2]\!]\varepsilon \\ & \qquad\qquad \mathtt{else}\ [\![e_2]\!]\varepsilon \end{array}$$

Now, from Proposition 13, we have $([\![e_1]\!]\varepsilon)[vs/\langle\!\langle x_1, x_2, x_3\rangle\!\rangle] = [\![e_1]\!]\varepsilon$ and $([\![e_2]\!]\varepsilon)[vs/\langle\!\langle x_1, x_2, x_3\rangle\!\rangle]$, for any $vs = \langle\!\langle v_1, v_2, v_3\rangle\!\rangle$, thus, we can use (43) once and (38) twice to get $([\![M]\!], [\![e]\!]\varepsilon) \to_0^* ([\![M]\!], [\![e_1]\!]\varepsilon)$, as required.

In the second subcase, where e_{b} is a value different from $\mathtt{true*}$, rule (5) must have been applied, thus, we have $e' = e_2$ and $M' = M$. Following similar reasoning as in the previous subcase, but with (39) used three times, we get $([\![M]\!], [\![e]\!]\varepsilon) \to_0^* ([\![M]\!], [\![e_2]\!]\varepsilon)$, as required.

In the third subcase where e_{b} is not a value, rule (1) must have been applied, thus, there exists e_{b}' such that $(M, e_{\mathrm{b}}) \hookrightarrow (M', e_{\mathrm{b}}')$ and $e' = \mathtt{if}\ e_{\mathrm{b}}'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$. By induction, we have $[\![(M, e_{\mathrm{b}})]\!]\varepsilon \to_0^* [\![(M'; e_{\mathrm{b}}')]\!]\varepsilon$. We also have

$$\begin{array}{rl} [\![e]\!]\varepsilon = & \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![e_{\mathrm{b}}]\!]\varepsilon \\ & \quad \mathtt{in}\ \mathtt{if}\ x_1 \triangleright \mathrm{r}\ \mathtt{then} \\ & \qquad\qquad \mathtt{if}\ x_2 \triangleright \mathrm{g}\ \mathtt{then}\ [\![e_1]\!]\varepsilon \\ & \qquad\qquad \mathtt{else\ if}\ x_3 \triangleright \mathrm{b}\ \mathtt{then}\ [\![e_1]\!]\varepsilon\ \mathtt{else}\ [\![e_2]\!]\varepsilon \\ & \qquad\quad \mathtt{else} \\ & \qquad\qquad \mathtt{if}\ x_2 \triangleright \mathrm{g}\ \mathtt{then\ if}\ x_3 \triangleright \mathrm{b}\ \mathtt{then}\ [\![e_1]\!]\varepsilon\ \mathtt{else}\ [\![e_2]\!]\varepsilon \\ & \qquad\qquad \mathtt{else}\ [\![e_2]\!]\varepsilon \end{array}$$

Because

$$\begin{array}{rl} [\![e']\!]\varepsilon = & \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![e_{\mathrm{b}}']\!]\varepsilon \\ & \quad \mathtt{in}\ \mathtt{if}\ x_1 \triangleright \mathrm{r}\ \mathtt{then} \\ & \qquad\qquad \mathtt{if}\ x_2 \triangleright \mathrm{g}\ \mathtt{then}\ [\![e_1]\!]\varepsilon \\ & \qquad\qquad \mathtt{else\ if}\ x_3 \triangleright \mathrm{b}\ \mathtt{then}\ [\![e_1]\!]\varepsilon\ \mathtt{else}\ [\![e_2]\!]\varepsilon \\ & \qquad\quad \mathtt{else} \\ & \qquad\qquad \mathtt{if}\ x_2 \triangleright \mathrm{g}\ \mathtt{then\ if}\ x_3 \triangleright \mathrm{b}\ \mathtt{then}\ [\![e_1]\!]\varepsilon\ \mathtt{else}\ [\![e_2]\!]\varepsilon \\ & \qquad\qquad \mathtt{else}\ [\![e_2]\!]\varepsilon \end{array}$$

we can apply Proposition 11 to get $[\![(M, e)]\!]\varepsilon \to_0^* [\![(M'; e')]\!]\varepsilon$, as required.

$\boxed{\textsc{Case}\ e = \lambda x.e_1}$ It follows by inspection that (2) must have been applied, thus, we have $e' = l$ and $M' = (M, l \mapsto \lambda x.e_1)$. From the definition of translation, we have $[\![e]\!]\varepsilon = \lambda^{\langle \mathrm{r,g,b}\rangle}\langle\!\langle x_1, x_2, x_3\rangle\!\rangle.[\![e_1]\!](x \mapsto (x_1, x_2, x_3))$. From (36), we have $[\![(M; e)]\!]\varepsilon \to_0 \Big(([\![M]\!], l \mapsto \lambda\langle\!\langle x_1, x_2, x_3\rangle\!\rangle.[\![e_1]\!](x \mapsto (x_1, x_2, x_3))); \langle\!\langle l, l, l\rangle\!\rangle\Big)$. It follows from the definition of M' and e' and from the definition of translation of machine states that $[\![(M; e)]\!]\varepsilon \to_0 [\![(M'; e')]\!]\varepsilon$. Now from (48), we have $[\![(M; e)]\!]\varepsilon \to_0^* [\![(M'; e')]\!]\varepsilon$, as required.

$\boxed{\textsc{Case}\ e = e_1\ e_2}$ There are several subcases to consider. Either $e_1 = l$ and $e_2 = v$, in which case (3) is applied, or $e_1 = l$ and e_2 is a non-value expression, in which case (1) is applied, or both e_1 and e_2 are non-value expressions, in which case (1) is also applied.

We first consider the case where (3) is applied. In this case $M(l) = \lambda x.e_0$ and $e' = e_0[v/x]$ and $M' = M$. From the definition of translation, we have

$$\begin{array}{rl} [\![e]\!]\varepsilon = & \mathtt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = \langle\!\langle l, l, l\rangle\!\rangle \\ & \quad \mathtt{in}\ \mathtt{appe}\ [x_1 \triangleright \mathrm{r}, x_2 \triangleright \mathrm{g}]\ \langle\!\langle v, v, v\rangle\!\rangle \\ & \qquad\quad \mathtt{else}\ \mathtt{appe}\ [x_1 \triangleright \mathrm{r}, x_3 \triangleright \mathrm{b}]\ \langle\!\langle v, v, v\rangle\!\rangle \\ & \qquad\qquad\quad \mathtt{else}\ (x_2 \triangleright \mathrm{g})\ \langle\!\langle v, v, v\rangle\!\rangle \end{array}$$

Using (43), it follows that we have

$$\begin{array}{rl} [\![M; e]\!]\varepsilon \to_0 [\![M]\!]; & \mathtt{appe}\ [l \triangleright \mathrm{r}, l \triangleright \mathrm{g}]\ \langle\!\langle v, v, v\rangle\!\rangle \\ & \quad \mathtt{else}\ \mathtt{appe}\ [l \triangleright \mathrm{r}, l \triangleright \mathrm{b}]\ \langle\!\langle v, v, v\rangle\!\rangle \\ & \qquad\quad \mathtt{else}\ (l \triangleright \mathrm{g})\ \langle\!\langle v, v, v\rangle\!\rangle \end{array}$$

Now, it also follows that $[\![M]\!](l) = \lambda\langle\!\langle x_1, x_2, x_3\rangle\!\rangle.[\![e_0]\!](x \mapsto (x_1, x_2, x_3))$. Thus, using (44), (48), and (49), it follows that $[\![M; e]\!]\varepsilon \to_0^* [\![M]\!]; [\![e_0]\!](x \mapsto (x_1, x_2, x_3))[\langle\!\langle v, v, v\rangle\!\rangle/\langle\!\langle x_1, x_2, x_3\rangle\!\rangle]$. Finally, from Proposition 12 and because $e' = e_0[v/x]$, we can conclude $[\![M; e]\!]\varepsilon \to_0^* [\![M'; e']\!]\varepsilon$, as required.

The remaining subcases make use of contextual reasoning and are straightforward.

$\boxed{\text{CASE } e = e_1\ op\ e_2}$ The proof for this case is similar to the case for application.

□

Notice that Proposition 14 does not assume well-typedness of S and S'.

4.2 Fault Propagation

Before we can state a fault tolerance property for the system, we define a *fault propagation relation*, $S \succ_c S'$, where S and S' are machine states and c is a color. Provided S is a non-faulty machine state, S' denotes a machine state that may differ from S in that values decorated with the color c may differ in S and S'.

Machine State Fault Propagation $\boxed{S \succ_c S'}$

$$\frac{M \succ_c M' \quad e \succ_c e'}{(M; e) \succ_c (M'; e')} \tag{68}$$

Memory Fault Propagation $\boxed{M \succ_c M'}$

$$\frac{M \succ_c M' \quad e \succ_c e'}{(M, l \mapsto \lambda xs.e) \succ_c (M', l \mapsto \lambda xs.e')} \tag{69}$$

$$\frac{}{\varepsilon \succ_c \varepsilon} \tag{70}$$

Expression Fault Propagation $\boxed{e \succ_c e'}$

$$\frac{}{x \triangleright c' \succ_c x \triangleright c'} \tag{71}$$

$$\frac{}{v \triangleright c \succ_c v' \triangleright c} \tag{72}$$

$$\frac{}{w \succ_c w} \tag{73}$$

$$\frac{}{n \triangleright c' \succ_c n \triangleright c'} \tag{74}$$

$$\frac{}{\texttt{true} \triangleright c' \succ_c \texttt{true} \triangleright c'} \tag{75}$$

$$\frac{}{\texttt{false} \triangleright c' \succ_c \texttt{false} \triangleright c'} \tag{76}$$

$$\frac{e \succ_c e'}{\lambda^{cs} xs.e \succ_c \lambda^{cs} xs.e'} \tag{77}$$

$$\frac{e \succ_c e' \quad e_1 \succ_c e_1' \quad e_2 \succ_c e_2'}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \succ_c \texttt{if}\ e'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2'} \tag{78}$$

$$\frac{e_1 \succ_c e_1' \quad e_2 \succ_c e_2'}{\texttt{let}\ xs = e_1\ \texttt{in}\ e_2 \succ_c \texttt{let}\ xs = e_1'\ \texttt{in}\ e_2'} \tag{79}$$

$$\frac{e_i \succ_c e_i' \quad i = 1..n}{\langle e_1, \cdots, e_n \rangle \succ_c \langle e_1', \cdots, e_n' \rangle} \tag{80}$$

$$\frac{e_1 \succ_c e_1' \quad e_2 \succ_c e_2'}{e_1\ op_{c'}\ e_2 \succ_c e_1'\ op_{c'}\ e_2'} \tag{81}$$

$$\frac{e_1 \succ_c e_1' \quad e_2 \succ_c e_2' \quad e_3 \succ_c e_3' \quad e_4 \succ_c e_4'}{\texttt{appe}\ [e_1, e_2]\ e_3\ \texttt{else}\ e_4 \succ_c \texttt{appe}\ [e_1', e_2']\ e_3'\ \texttt{else}\ e_4'} \tag{82}$$

$$\frac{e_1 \succ_c e_1' \quad e_2 \succ_c e_2'}{e_1\ e_2 \succ_c e_1'\ e_2'} \tag{83}$$

Notice that only rule (72) is particularly interesting. The remaining rules specify that the relation is closed under pair-wise simple induction on subexpressions.

When S and S' are machine states, we write $S \succ S'$ to mean that there exists c such that $S \succ_c S'$. The relation $S \succ S'$ expresses that S' may differ from S "only on one color". Intuitively, after a transient fault, one of the colors have become faulty, but the remaining two colors can still be trusted. Notice that the relation $\succ$ is reflexive and symmetric (we shall not make use of the property that $\succ$ is symmetric).

The following proposition states that translation, substitution, and fault propagation interact in interesting ways:

PROPOSITION 15 (Fault Propagation Properties). *Let* $\rho' = \rho, x \mapsto (x_1, x_2, x_3)$ *and* $xs = \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle$.

1. $[\![e[v/x]]\!]\rho \succ_r [\![e]\!]\rho'[\langle\!\langle v', v, v \rangle\!\rangle/xs]$
2. $[\![e[v/x]]\!]\rho \succ_g [\![e]\!]\rho'[\langle\!\langle v, v', v \rangle\!\rangle/xs]$
3. $[\![e[v/x]]\!]\rho \succ_b [\![e]\!]\rho'[\langle\!\langle v, v, v' \rangle\!\rangle/xs]$

PROOF By induction over the structure e. □

We shall make use of yet another proposition, which states that once a color is considered faulty, more faults are allowed to be introduced on that color via substitutions:

PROPOSITION 16 (Fault Propagation Substitution). *Let* $\rho' = \rho, x \mapsto (x_1, x_2, x_3)$ *and* $xs = \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle$.

1. $[\![e]\!]\rho' \succ_r e' \Rightarrow [\![e[v/x]]\!]\rho \succ_r e'[\langle\!\langle v', v, v \rangle\!\rangle/xs]$
2. $[\![e]\!]\rho' \succ_g e' \Rightarrow [\![e[v/x]]\!]\rho \succ_g e'[\langle\!\langle v, v', v \rangle\!\rangle/xs]$
3. $[\![e]\!]\rho' \succ_b e' \Rightarrow [\![e[v/x]]\!]\rho \succ_b e'[\langle\!\langle v, v, v' \rangle\!\rangle/xs]$

PROOF By induction over the structure e. □

The proposition can be motivated by the following example, which shows that, due to substitutions, faults on a color may accumulate:

$$\begin{array}{lll} e & = & \texttt{let}\ \langle\!\langle y_1, y_2, y_3 \rangle\!\rangle = \langle\!\langle v, v, v' \rangle\!\rangle \\ & & \texttt{in let}\ \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle = \langle\!\langle y_1, y_2, y_3 \rangle\!\rangle \\ & & \quad \texttt{in}\ \langle\!\langle x_1 + y_1, x_2 + y_2, x_3 + y_3 \rangle\!\rangle \\ \rightarrow_0 & & \texttt{let}\ \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle = \langle\!\langle v, v, v' \rangle\!\rangle \\ & & \texttt{in}\ \langle\!\langle x_1 + v, x_2 + v, x_3 + v' \rangle\!\rangle \\ \rightarrow_0 & & \langle\!\langle v + v, v + v, v' + v' \rangle\!\rangle \end{array}$$

Here e is the initial expression in which a faulty value v' is being bound to y_3 and the non-faulty copies of the value v is being bound to y_1 and y_2. By substitution, the single faulty-value is propagated to two different places. However, the notion of substitution disallows faults to propagate across colors.

The fault propagation relation is extended to evaluation contexts according to the rules below.

Evaluation Context Fault Propagation $\boxed{E \succ_c E'}$

$$\frac{}{[\cdot] \succ_c [\cdot]} \tag{84}$$

$$\frac{E \succ_c E' \quad e \succ_c e'}{E\ e \succ_c E'\ e'} \tag{85}$$

$$\frac{w \succ_c w' \quad E \succ_c E'}{w\ E \succ_c w'\ E'} \tag{86}$$

$$\frac{E_1 \succ_c E_2 \quad e_1 \succ_c e_2 \quad e_1' \succ_c e_2' \quad e_1'' \succ_c e_2''}{\texttt{appe}\ [E_1, e_1]\ e_1'\ \texttt{else}\ e_1'' \succ_c \texttt{appe}\ [E_2, e_2]\ e_2'\ \texttt{else}\ e_2''} \tag{87}$$

$$\frac{w_1 \succ_c w_2 \quad E_1 \succ_c E_2 \quad e_1 \succ_c e_2 \quad e_1' \succ_c e_2'}{\texttt{appe}\ [w_1, E_1]\ e_1\ \texttt{else}\ e_1' \succ_c \texttt{appe}\ [w_2, E_2]\ e_2\ \texttt{else}\ e_2'} \tag{88}$$

$$\frac{w_1 \succ_c w_2 \quad w_1' \succ_c w_2' \quad E_1 \succ_c E_2 \quad e_1 \succ_c e_2}{\texttt{appe}\ [w_1, w_1']\ E_1\ \texttt{else}\ e_1 \succ_c \texttt{appe}\ [w_2, w_2']\ E_2\ \texttt{else}\ e_2} \tag{89}$$

$$\frac{E \succ_c E' \quad e \succ_c e'}{\texttt{let}\ \mathit{xs} = E\ \texttt{in}\ e \succ_c \texttt{let}\ \mathit{xs} = E'\ \texttt{in}\ e'} \tag{90}$$

$$\frac{E \succ_c E' \quad e_1 \succ_c e_1' \quad e_2 \succ_c e_2'}{\texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \succ_c \texttt{if}\ E'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2'} \tag{91}$$

$$\frac{E \succ_c E' \quad e \succ_c e'}{E\ op_c\ e \succ_c E'\ op_{c'}\ e'} \tag{92}$$

$$\frac{w \succ_c w' \quad E \succ_c E'}{w\ op_c\ E \succ_c w'\ op_{c'}\ E'} \tag{93}$$

$$\frac{\begin{array}{c} w_i \succ_c w_i',\ \ i = [1..n] \\ E \succ_c E' \quad e_j \succ_c e_j',\ \ j = [1..m] \end{array}}{\langle w_1, .., w_n, E, e_1, .., e_m\rangle \succ_c \langle w_1', .., w_n', E', e_1', .., e_m'\rangle} \tag{94}$$

The following proposition expresses that the results of filling contexts is related by the fault propagation relation, provided the contexts and the filled-in objects are related:

PROPOSITION 17 (Fault Propagation Context Filling). *If $E \succ_c E'$ and $S \succ_c S'$ then $E[S] \succ_c E'[S']$.*

PROOF By induction on the derivation $E \succ_c E'$. □

The following proposition states that evaluation contexts are preserved by the fault propagation relation.

PROPOSITION 18 (Fault Propagation Context Preservation). *If $E[S] \succ_c S'$ then there exists E' and S'' such that $S \succ_c S''$ and $E \succ_c E'$ and $S' = E'[S'']$.*

PROOF By induction on the structure of E. □

The following proposition states that the translation is compositional with respect to context filling.

PROPOSITION 19 (Translation is Contextual Compositional). *If $S = [\![E[S_s]]\!]\varepsilon$ and $S' = [\![E[S_s']]\!]\varepsilon$ then there exists E' such that $S = E'[[\![S_s]\!]\varepsilon]$ and $S' = E'[[\![S_s']\!]\varepsilon]$.*

PROOF By induction on the structure of E. We give the proof for one case below.

$\boxed{\text{CASE } E = \texttt{let}\ x = E_0\ \texttt{in}\ e_0}$ Let $S = (M; e)$ and $S' = (M'; e')$. From the definition of translation, we have

$$\begin{array}{rcl} [\![E[S]]\!]\varepsilon & = & ([\![M]\!]; [\![\texttt{let}\ x = E_0[e]\ \texttt{in}\ e_0]\!]\varepsilon) \\ & = & ([\![M]\!];\ \texttt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![E_0[e]]\!]\varepsilon\ \ ) \\ & & \qquad\quad \texttt{in}\ [\![e_0]\!](x \mapsto (x_1, x_2, x_3)) \\ [\![E[S']]\!]\varepsilon & = & ([\![M']\!]; [\![\texttt{let}\ x = E_0[e']\ \texttt{in}\ e_0]\!]\varepsilon) \\ & = & ([\![M']\!];\ \texttt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![E_0[e']]\!]\varepsilon\ \ ) \\ & & \qquad\quad \texttt{in}\ [\![e_0]\!](x \mapsto (x_1, x_2, x_3)) \end{array}$$

Let $S_0 = E_0[S]$ and $S_0' = E_0[S']$. By induction, there exists E_0' such that

$$[\![E_0[S]]\!]\varepsilon = E_0'[[\![S]\!]\varepsilon] \tag{95}$$

$$[\![E_0[S']]\!]\varepsilon = E_0'[[\![S']\!]\varepsilon] \tag{96}$$

Let

$$\begin{array}{rcl} E' & = & \texttt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = E_0' \\ & & \texttt{in}\ [\![e_0]\!](x \mapsto (x_1, x_2, x_3)) \end{array} \tag{97}$$

Using (95) and (97), we have

$$\begin{array}{rcl} [\![E[S]]\!]\varepsilon & = & ([\![M]\!]; [\![\texttt{let}\ x = E_0[e]\ \texttt{in}\ e_0]\!]\varepsilon) \\ & = & ([\![M]\!];\ \texttt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = [\![E_0[e]]\!]\varepsilon\ \ ) \\ & & \qquad\quad \texttt{in}\ [\![e_0]\!](x \mapsto (x_1, x_2, x_3)) \\ & = & ([\![M]\!];\ \texttt{let}\ \langle\!\langle x_1, x_2, x_3\rangle\!\rangle = E_0'[[\![e]\!]\varepsilon]\ \ ) \\ & & \qquad\quad \texttt{in}\ [\![e_0]\!](x \mapsto (x_1, x_2, x_3)) \\ & = & ([\![M]\!]; E'[[\![e]\!]\varepsilon]) \\ & = & E'[[\![S]\!]\varepsilon] \end{array}$$

as required. Similarly, using (96), we can also, as required, derive $[\![E[S']]\!]\varepsilon = E'[[\![S']\!]\varepsilon]$ □

4.3 Single Step Fault Tolerance

The following proposition states that whenever a transient fault occurs during execution of an extended-language program, the resulting machine state is related (via the fault propagation relation) to the corresponding non-faulty machine state:

PROPOSITION 20 (Faults Are Captured by Fault Propagation). *If $S \to_1 S_0^{z}$ then $S \succ S^{z}$.*

PROOF By straightforward induction over E, using (47). □

Now, consider again the diagram from Section 1:

$$\begin{tikzcd} S \arrow[rr, hook] \arrow[d, Rightarrow] & & S' \arrow[d, Rightarrow] \\ S_0 \arrow[r, "*_0"] & S_i \arrow[r, "*_0"] \arrow[d] & S_n \arrow[d, phantom, "\curlyvee"] \\ & S^{z} \arrow[r, "*_0"] & S_n^{z} \end{tikzcd}$$

We have already covered the upper part of this diagram with Proposition 14, which states that a translated program simulates the source language program. We are now in the position to express the heart of the fault tolerance property, which allows the target-level program to tolerate a single transient fault during execution.

The proposition expresses that the fault propagation relation on machine states is closed under evaluation. In terms of the machine states in the diagram, if a fault occurs at machine state S_i, enforcing the program into a machine state S^{z}, then, assuming no additional faults, S^{z} may evaluate into a machine state S_n^{z}, which is related to the source language machine state S' via voting.

PROPOSITION 21 (Single Step Fault Tolerance). *Assume $S \hookrightarrow S'$ and $[\![S]\!]\varepsilon \to_0^* S_0 \to_0^* [\![S']\!]\varepsilon$.*

For any S_0^{z} and color c such that $S_0 \succ_c S_0^{z}$ there exists S^{z} such that $S_0^{z} \to_0^* S^{z}$ and $[\![S']\!]\varepsilon \succ_c S^{z}$.

PROOF By induction over the derivation $S \hookrightarrow S'$, where $S = (M; e_0)$.

$\boxed{\text{CASE } S = (M; \texttt{true})}$ For this case we know (7) is applied. We have $[\![S]\!]\varepsilon = ([\![M]\!]; e)$, where $e = \langle\!\langle \texttt{true}, \texttt{true}, \texttt{true}\rangle\!\rangle$

and $S' = (M; \texttt{true*})$ and $\llbracket S' \rrbracket \varepsilon = (\llbracket M \rrbracket; e')$, where $e' = \langle\!\langle \texttt{true*}, \texttt{true*}, \texttt{true*} \rangle\!\rangle$. We also have

$$\llbracket S \rrbracket \varepsilon \rightarrow_0 S_1 \rightarrow_0 S_2 \rightarrow_0 \llbracket S' \rrbracket \varepsilon$$

where

$$\begin{array}{lll} S_1 = (M; e_1) & \text{and} & e_1 = \langle\!\langle \texttt{true*}, \texttt{true}, \texttt{true} \rangle\!\rangle \\ S_2 = (M; e_2) & \text{and} & e_2 = \langle\!\langle \texttt{true*}, \texttt{true*}, \texttt{true} \rangle\!\rangle \end{array}$$

There are now four subcases to consider:

$\underline{\text{SUBCASE } S_0 = \llbracket S \rrbracket \varepsilon :}$ From the definition of $\succ_c$, we have $S_0^{\text{z}} = (M^{\text{z}}; e)$ — there are no values to zap in e. Let $S_1^{\text{z}} = (M^{\text{z}}; e')$. We then have from repeated use of (46), (41), and (48) that $S_0^{\text{z}} \rightarrow_0^* S_1^{\text{z}}$. Moreover, from the definition of $\succ$ it follows trivially that $\llbracket S' \rrbracket \varepsilon \succ S_1^{\text{z}}$, as required.

$\underline{\text{SUBCASE } S_0 = S_1 :}$ From the definition of $\succ_c$, we have $\llbracket M \rrbracket \succ_c M_0^{\text{z}}$ and $e_1 \succ_c e_0^{\text{z}}$, where $S_0^{\text{z}} = (M_0^{\text{z}}; e_0^{\text{z}})$. We now proceed by case analysis of c.

First consider the case where $c = \text{r}$. From the definition of $\succ_c$, we have $e_0^{\text{z}} = \langle\!\langle v, \texttt{true}, \texttt{true} \rangle\!\rangle$ for some v. It follows that we have $S_0^{\text{z}} \rightarrow_0^* S_1^{\text{z}}$, where $S_1^{\text{z}} = (M_0^{\text{z}}; e_1^{\text{z}})$ and $e_1^{\text{z}} = \langle\!\langle v, \texttt{true*}, \texttt{true} \rangle\!\rangle$. Moreover, $\llbracket S' \rrbracket \varepsilon \succ_c S_1^{\text{z}}$, as required.

Alternatively, we have $c = \text{g}$ or $c = \text{b}$. From the definition of $\succ_c$, we then have $e_0^{\text{z}} = e_1$. It follows that $S_0^{\text{z}} \rightarrow_0^* S_1^{\text{z}}$, where $S_1^{\text{z}} = (M_0^{\text{z}}; e')$. Moreover, from the definition of $\succ_c$, we have $\llbracket S' \rrbracket \varepsilon \succ_c S_1^{\text{z}}$, as required.

$\underline{\text{SUBCASE } S_0 = S_2 :}$ From the definition of $\succ_c$, we have $\llbracket M \rrbracket \succ_c M_0^{\text{z}}$ and $e_2 \succ_c e_0^{\text{z}}$, where $S_0^{\text{z}} = (M_0^{\text{z}}; e_0^{\text{z}})$. We now proceed by case analysis of c.

First assume $c = \text{r}$. From the definition of $\succ_c$, we have $e_0^{\text{z}} = \langle\!\langle v, \texttt{true*}, \texttt{true} \rangle\!\rangle$ for some v. It follows that we have $S_0^{\text{z}} \rightarrow_0^* S_1^{\text{z}}$, where $S_1^{\text{z}} = (M_0^{\text{z}}; e_1^{\text{z}})$ and $e_1^{\text{z}} = \langle\!\langle v, \texttt{true*}, \texttt{true*} \rangle\!\rangle$. Moreover, $\llbracket S' \rrbracket \varepsilon \succ_c S_1^{\text{z}}$ holds, as required.

Second, assume $c = \text{g}$. The result follows similarly as for the case $c = \text{r}$, but with $e_0^{\text{z}} = \langle\!\langle \texttt{true*}, v, \texttt{true*} \rangle\!\rangle$ for some v and $e_1^{\text{z}} = \langle\!\langle \texttt{true*}, v, \texttt{true*} \rangle\!\rangle$.

Finally, assume $c = \text{b}$. From the definition of $\succ_c$, we have $e_0^{\text{z}} = e_2$. It follows that $S_0^{\text{z}} \rightarrow_0^* S_1^{\text{z}}$, where $S_1^{\text{z}} = (M_0^{\text{z}}; e')$. Moreover, from the definition of $\succ_c$, we have $\llbracket S' \rrbracket \varepsilon \succ_c S_1^{\text{z}}$, as required.

$\underline{\text{SUBCASE } S_0 = \llbracket S' \rrbracket \varepsilon :}$ Trivially true with $S_1^{\text{z}} = S_0^{\text{z}}$.

$\boxed{\text{CASE } S = (M; \texttt{let } x \texttt{ = } v \texttt{ in } e_s)}$ For this case we know (9) is applied. Let $xs = \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle$ and $vs = \langle\!\langle v, v, v \rangle\!\rangle$. We have $\llbracket S \rrbracket \varepsilon = (\llbracket M \rrbracket; e)$, where

$$\begin{array}{rcll} e & = & \texttt{let } xs \texttt{ = } vs \texttt{ in } \llbracket e_{\text{s}} \rrbracket (x \mapsto (x_1, x_2, x_3)) & \\ S' & = & (M; e_{\text{s}}[v/x]) & \\ \llbracket S' \rrbracket \varepsilon & = & (\llbracket M \rrbracket; e') & (98) \\ e' & = & \llbracket e_{\text{s}}[v/x] \rrbracket \varepsilon & (99) \end{array}$$

Using Proposition 12, we also have

$$\begin{array}{rl} \llbracket S \rrbracket \varepsilon \quad \rightarrow_0 & (\llbracket M \rrbracket; \llbracket e_{\text{s}} \rrbracket (x \mapsto (x_1, x_2, x_3))[vs/xs] \\ & = (\llbracket M \rrbracket; e') = \llbracket S' \rrbracket \varepsilon \end{array}$$

Thus there are two subcases to consider.

$\underline{\text{SUBCASE } S_0 = \llbracket S \rrbracket \varepsilon :}$ From the definition of $\succ_c$, we have

$$\llbracket M \rrbracket \quad \succ_c \quad M_0^{\text{z}} \qquad (100)$$

and $e \succ_c e_0^{\text{z}}$ and $S_0 = (M_0^{\text{z}}; e_0^{\text{z}})$. We now proceed by case analysis of c.

First, consider the case $c = \text{r}$. From the definition of $\succ_{\text{r}}$, we have

$$e_0^{\text{z}} = \texttt{let } xs \texttt{ = } vs' \texttt{ in } e_{\text{s}}^{\text{z}}$$

where

$$\llbracket e_{\text{s}} \rrbracket (x \mapsto (x_1, x_2, x_3)) \quad \succ_{\text{r}} \quad e_{\text{s}}^{\text{z}} \qquad (101)$$

and $vs' = \langle\!\langle v', v, v \rangle\!\rangle$ for arbitrary v'. From (43), there exists S_1^{z} such that

$$S_0^{\text{z}} \quad \rightarrow_0 \quad S_1^{\text{z}} \qquad (102)$$

where $S_1^{\text{z}} = (M_0^{\text{z}}; e_1^{\text{z}})$ and $e_1^{\text{z}} = e_{\text{s}}^{\text{z}}[vs'/xs]$. From Proposition 16(1) and (101), we have $\llbracket e_{\text{s}}[v/x] \rrbracket \varepsilon \succ_{\text{r}} e_1^{\text{z}}$. Thus, from (99), we have

$$e' \quad \succ_{\text{r}} \quad e_1^{\text{z}} \qquad (103)$$

From the definition of $\succ_c$ and from (103), (98), and (100), we have $\llbracket S' \rrbracket \varepsilon \succ_c S_1^{\text{z}}$. Moreover, from (102) and (48), we also have $S_0^{\text{z}} \rightarrow_0^* S_1^{\text{z}}$, as required.

The cases for $c = \text{g}$ and $c = \text{b}$ are similar to the case for $c = \text{r}$.

$\underline{\text{SUBCASE } S_0 = \llbracket S' \rrbracket \varepsilon :}$ Follows trivially with $S_1^{\text{z}} = S_0^{\text{z}}$ using (49).

$\boxed{\text{CASE } S = (M; l\ v)}$ For this case, we know (3) is applied. Let $xs = \langle\!\langle x_1, x_2, x_3 \rangle\!\rangle$. We have $\llbracket S \rrbracket \varepsilon = (\llbracket M \rrbracket; e)$, where

```
e = let xs = ⟦l⟧ε
    in appe [x₁ ▷ r, x₂ ▷ g] (⟦v⟧ε)
       else appe [x₁ ▷ r, x₃ ▷ b] (⟦v⟧ε)
            else (x₂ ▷ g) (⟦v⟧ε)
```

and $M(l) = \lambda x.e^0$. From (43), we have $(\llbracket M \rrbracket; e) \rightarrow_0 S_1$, where

$$S_1 \quad = \quad (\llbracket M \rrbracket; e_1) \qquad (104)$$

and

```
e₁ = appe [l ▷ r, l ▷ g] (⟦v⟧ε)
     else appe [l ▷ r, l ▷ b] (⟦v⟧ε)
          else (l ▷ g) (⟦v⟧ε)
```

Moreover, from (44), we have $S_1 \rightarrow_0 S_2$, where $S_2 = (\llbracket M \rrbracket; e_2)$ and

$$\begin{array}{rcll} e_2 & = & \llbracket e^0 \rrbracket (x \mapsto (x_1, x_2, x_3))[vs/xs] & (105) \\ \llbracket M \rrbracket (l) & = & \lambda xs. \llbracket e^0 \rrbracket (x \mapsto (x_1, x_2, x_3)) & (106) \\ vs & = & \langle\!\langle v, v, v \rangle\!\rangle & \end{array}$$

We also have from (3) that

$$S' \quad = \quad (M; e^0[v/x]) \qquad (107)$$

Using Proposition 16(1), we have $\llbracket S' \rrbracket \varepsilon = S_2$. Thus, we have

$$\llbracket S \rrbracket \varepsilon \rightarrow_0 S_1 \rightarrow_0 \llbracket S' \rrbracket \varepsilon$$

which leads to three subcases to consider for S_0, namely $S_0 = \llbracket S \rrbracket \varepsilon$ or $S_0 = S_1$ or $S_0 = \llbracket S' \rrbracket \varepsilon$.

$\underline{\text{SUBCASE } S_0 = \llbracket S \rrbracket \varepsilon :}$ From the definition of $\succ_c$, we have $S_0^{\text{z}} = (M_0^{\text{z}}; e_0^{\text{z}})$, where

$$\begin{array}{rcll} \llbracket M \rrbracket & \succ_c & M_0^{\text{z}} & (108) \\ e & \succ_c & e_0^{\text{z}} & (109) \end{array}$$

We proceed by case analysis of c.

[SUBSUBCASE $c = \text{r}$:] From the definition of $\succ_{\text{r}}$ and from (109), we have

```
e₀ᶻ = let xs = ⟨⟨v₁, l, l⟩⟩
      in appe [x₁ ▷ r, x₂ ▷ g] ⟨⟨v₂, v, v⟩⟩
         else appe [x₁ ▷ r, x₃ ▷ b] ⟨⟨v₃, v, v⟩⟩
              else (x₂ ▷ g) ⟨⟨v₄, v, v⟩⟩
```

for arbitrary v_1, v_2, v_3, and v_4. Now, from (43), we have

$$S_0^z \rightarrow_0 S_1^z \quad (110)$$
$$S_1^z = (M_0^z; e_1^z) \quad (111)$$

and

$$e_1^z = \texttt{appe}\ [v_1 \triangleright \mathrm{r}, l \triangleright \mathrm{g}]\ \langle\!\langle v_2, v, v\rangle\!\rangle \quad (112)$$
$$\texttt{else appe}\ [v_1 \triangleright \mathrm{r}, l \triangleright \mathrm{b}]\ \langle\!\langle v_3, v, v\rangle\!\rangle$$
$$\texttt{else}\ (l \triangleright \mathrm{g})\ \langle\!\langle v_4, v, v\rangle\!\rangle$$

From (106), (108), and from the definition of $\succ_c$, we have there exists e_z^0 such that

$$M_0^z(l) = \lambda xs.e_z^0 \quad (113)$$
$$[\![e^0]\!](x \mapsto (x_1, x_2, x_3)) \succ_{\mathrm{r}} e_z^0 \quad (114)$$

There are now two cases; either $v_1 = l$ or $v_1 \neq l$.

We first consider the case for $v_1 = l$. From (111), (112), (113), and (44), we have

$$S_1^z \rightarrow_0 S_2^z \quad (115)$$
$$S_2^z = (M_0^z; e_2^z) \quad (116)$$
$$e_2^z = e_z^0[\langle\!\langle v_2, v, v\rangle\!\rangle / xs] \quad (117)$$

Now, from Proposition 16, (114), and (117), we have

$$[\![e^0[v/x]]\!]\varepsilon \succ_{\mathrm{r}} e_2^z \quad (118)$$

From (107), (118), (115), (116), (110), (108), and (48), we have there exists $S^z = S_2^z$ such that $S_0^z \rightarrow_0^* S^z$ and $[\![S']\!]\varepsilon \succ_c S^z$, as required.

We now consider the case for $v_1 \neq l$. From (112), (111), and from using (45) twice, we have

$$S_1^z \rightarrow_0 S_2^z \rightarrow_0 S_3^z \quad (119)$$
$$S_3^z = (M_0^z; e_3^z) \quad (120)$$
$$e_3^z = (l \triangleright \mathrm{g})\ \langle\!\langle v_4, v, v\rangle\!\rangle \quad (121)$$

From (120), (121), (113), and from (37), we have

$$S_3^z \rightarrow_0 S_4^z \quad (122)$$
$$S_4^z = (M_0^z; e_4^z)$$
$$e_4^z = e_z^0[\langle\!\langle v_4, v, v\rangle\!\rangle / xs] \quad (123)$$

Now, from Proposition 16, (114), and (123), we have

$$[\![e^0[v/x]]\!]\varepsilon \succ_{\mathrm{r}} e_4^z \quad (124)$$

From (107), (124), (110), (119), (122), (108), and (48), we have there exists $S^z = S_4^z$ such that $S_0^z \rightarrow_0^* S^z$ and $[\![S']\!]\varepsilon \succ_c S^z$, as required.

[SUBSUBCASE $c = \mathrm{g}$ or $c = \mathrm{b}$:] These cases are similar to the case for $c = \mathrm{r}$.

SUBCASE $S_0 = S_1$: From the definition of $\succ_c$ and from (104), we have $S_0^z = (M_0^z; e_0^z)$, where

$$[\![M]\!] \succ_c M_0^z \quad (125)$$
$$e_1 \succ_c e_0^z \quad (126)$$

We now proceed by case analysis of c.

[SUBSUBCASE $c = \mathrm{r}$:] From the definition of $\succ_{\mathrm{r}}$, we have

$$e_0^z = \texttt{appe}\ [v_1 \triangleright \mathrm{r}, l \triangleright \mathrm{g}]\ \langle\!\langle v_2, v, v\rangle\!\rangle$$
$$\texttt{else appe}\ [v_3 \triangleright \mathrm{r}, l \triangleright \mathrm{b}]\ \langle\!\langle v_4, v, v\rangle\!\rangle$$
$$\texttt{else}\ (l \triangleright \mathrm{g})\ \langle\!\langle v_5, v, v\rangle\!\rangle$$

for arbitrary v_1, v_2, v_3, v_4, and v_5. The remainder of this case follows the lines of the subcase $S_0 = [\![S]\!]\varepsilon$ with independent treatment of cases for whether $v_1 = l$ and whether $v_3 = l$.

[SUBSUBCASE $c = \mathrm{g}$ or $c = \mathrm{b}$:] These cases are similar to the case for $c = \mathrm{r}$.

SUBCASE $S_0 = [\![S']\!]\varepsilon$: The proof for this case follows immediately from reflexivity of $\rightarrow_0^*$ and $\succ_c$.

$\boxed{\text{CASE } S = (M; \lambda x.e^0)}$ For this case we know (2) is applied. We have $[\![S]\!]\varepsilon = ([\![M]\!]; e)$, where

$$e = \lambda^{\langle \mathrm{r},\mathrm{g},\mathrm{b}\rangle} xs.[\![e^0]\!](x \mapsto (x_1, x_2, x_3))$$

and $xs = \langle\!\langle x_1, x_2, x_3\rangle\!\rangle$. From (36), we have $[\![S]\!]\varepsilon \rightarrow_0 (M'; \langle\!\langle l, l, l\rangle\!\rangle)$, where

$$M' = [\![M]\!], l \mapsto \lambda xs.[\![e^0]\!](x \mapsto (x_1, x_2, x_3)) \quad (127)$$

From assumptions and (2), we have $S' = (M, l \mapsto \lambda x.e^0; l)$. It follows from the definition of translation of memory maps that

$$[\![S']\!]\varepsilon = (M'; \langle\!\langle l, l, l\rangle\!\rangle) \quad (128)$$

thus, we have $[\![S]\!]\varepsilon \rightarrow_0 [\![S']\!]\varepsilon$, which means that there are two cases to consider for S_0, namely $S_0 = [\![S]\!]\varepsilon$ and $S_0 = [\![S']\!]\varepsilon$.

SUBCASE $S_0 = [\![S]\!]\varepsilon$: From the definition of $\succ_c$, we have $S_0 \succ_c S^z$, where

$$S_0^z = (M_0^z; e_0^z) \quad (129)$$
$$[\![M]\!] \succ_c M_0^z \quad (130)$$
$$e_0^z = \lambda^{\langle \mathrm{r},\mathrm{g},\mathrm{b}\rangle} xs.e_z^0 \quad (131)$$
$$[\![e^0]\!](x \mapsto (x_1, x_2, x_3)) \succ_c e_z^0 \quad (132)$$

From (36), (129), and (131), we have

$$S_0^z \rightarrow_0 (M_1^z; e_1^z) \quad (133)$$

where

$$M_1^z = M_0^z, l \mapsto \lambda xs.e_z^0 \quad (134)$$
$$e_1^z = \langle\!\langle l, l, l\rangle\!\rangle \quad (135)$$

From the definition of $\succ_c$ and from the definition of translation and from (130), (134), (127), and (132), we have

$$M' \succ_c M_1^z \quad (136)$$

Moreover, from reflexivity of $\succ_c$, from (48), and from (136) and (128), there exists $S^z = (M_1^z; e_1^z)$ such that $S_0^z \rightarrow_0^* S^z$ and $[\![S']\!]\varepsilon \succ_c S^z$, as required.

SUBCASE $S_0 = [\![S']\!]\varepsilon$: The proof for this subcase follows immediately from reflexivity of $\rightarrow_0^*$ and $\succ_c$.

$\boxed{\text{CASE } S = (M; \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2)}$ The proof for this case is similar to, but simpler than, the case for application.

$\boxed{\text{CASE } S = (M; n_1\texttt{*} \texttt{+} n_2\texttt{*})}$ For this case, we know that (10) is applied, thus, we have

$$S' = (M'; n\texttt{*}) \quad (137)$$
$$n = n_1 + n_2 \quad (138)$$

From the definition of translation, we have

$$[\![S]\!]\varepsilon = ([\![M]\!]; e) \quad (139)$$
$$e = \texttt{let}\ xs = \langle\!\langle n_1\texttt{*}, n_1\texttt{*}, n_1\texttt{*}\rangle\!\rangle \quad (140)$$
$$\texttt{in let}\ ys = \langle\!\langle n_2\texttt{*}, n_2\texttt{*}, n_2\texttt{*}\rangle\!\rangle$$
$$\texttt{in}\ \langle\ (x_1 \triangleright \mathrm{r}) \texttt{+}_{\mathrm{r}} (y_1 \triangleright \mathrm{r}),$$
$$(x_2 \triangleright \mathrm{g}) \texttt{+}_{\mathrm{g}} (y_2 \triangleright \mathrm{g}),$$
$$(x_3 \triangleright \mathrm{b}) \texttt{+}_{\mathrm{b}} (y_3 \triangleright \mathrm{b})\ \rangle$$
$$xs = \langle\!\langle x_1, x_2, x_3\rangle\!\rangle$$
$$ys = \langle\!\langle y_1, y_2, y_3\rangle\!\rangle$$

for distinct choices of x_1, x_2, x_3, y_1, y_2, and y_3.

From (43), (46), (31), and from (139), (140), (137), and (138), we have

$$\llbracket S \rrbracket\varepsilon \rightarrow_0 S_1 \rightarrow_0 S_2 \rightarrow_0 S_3 \rightarrow_0 S_4 \rightarrow_0 \llbracket S' \rrbracket\varepsilon \tag{141}$$

where $S_1 = (\llbracket M \rrbracket; e_1)$ and $S_2 = (\llbracket M \rrbracket; e_2)$ and $S_3 = (\llbracket M \rrbracket; e_3)$ and $S_4 = (\llbracket M \rrbracket; e_4)$ and $\llbracket S' \rrbracket\varepsilon = (\llbracket M \rrbracket; \llbracket n* \rrbracket\varepsilon)$ and

$$\begin{aligned} e_1 &= \texttt{let}\ ys = \langle\!\langle n_2*, n_2*, n_2* \rangle\!\rangle \\ &\quad\ \texttt{in}\ \langle\ (n_1* \triangleright \mathrm{r}) +_{\mathrm{r}} (y_1 \triangleright \mathrm{r}), \\ &\qquad (n_1* \triangleright \mathrm{g}) +_{\mathrm{g}} (y_2 \triangleright \mathrm{g}), \\ &\qquad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (y_3 \triangleright \mathrm{b})\ \rangle \end{aligned} \tag{142}$$

$$\begin{aligned} e_2 &= \langle\ (n_1* \triangleright \mathrm{r}) +_{\mathrm{r}} (n_2* \triangleright \mathrm{r}), \\ &\quad (n_1* \triangleright \mathrm{g}) +_{\mathrm{g}} (n_2* \triangleright \mathrm{g}), \\ &\quad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (n_2* \triangleright \mathrm{b})\ \rangle \end{aligned} \tag{143}$$

$$\begin{aligned} e_3 &= \langle\ n*, \\ &\quad (n_1* \triangleright \mathrm{g}) +_{\mathrm{g}} (n_2* \triangleright \mathrm{g}), \\ &\quad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (n_2* \triangleright \mathrm{b})\ \rangle \end{aligned} \tag{144}$$

$$\begin{aligned} e_4 &= \langle\ n*, \\ &\quad n*, \\ &\quad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (n_2* \triangleright \mathrm{b})\ \rangle \end{aligned} \tag{145}$$

Thus, there are six subcases to check, namely $S_0 = \llbracket S \rrbracket\varepsilon$, $S_0 = S_1$, $S_0 = S_2$, $S_0 = S_3$, $S_0 = S_4$, and $S_0 = \llbracket S' \rrbracket\varepsilon$. The case for $S_0 = \llbracket S' \rrbracket\varepsilon$ follows trivially due to reflexivity of $\succ_c$. We show one of the remaining proof cases.

SUBCASE $S_0 = S_3$: From the definition of $\succ_c$, we have

$$\begin{aligned} S^{\mathrm{z}}_0 &= (M^{\mathrm{z}}_0; e^{\mathrm{z}}_0) && (146) \\ \llbracket M \rrbracket &\succ_c M^{\mathrm{z}}_0 && (147) \\ e_3 &\succ_c e^{\mathrm{z}}_0 && (148) \end{aligned}$$

We now proceed by case analysis of c and show the case for $c = \mathrm{g}$ below; the remaining cases are similar.

[SUBSUBCASE $c = \mathrm{g}$:] It follows that

$$\begin{aligned} e^{\mathrm{z}}_0 &= \langle\ n* \triangleright \mathrm{r}, \\ &\quad (v_2 \triangleright \mathrm{g}) +_{\mathrm{g}} (v_2 \triangleright \mathrm{g}), \\ &\quad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (n_2* \triangleright \mathrm{b})\ \rangle \end{aligned} \tag{149}$$

for arbitrary v_1 and v_2. There are now two cases; either v_1 and v_2 are both numbers or one of v_1 and v_2 is not a number.

Case $v_1 = n'_1*$ and $v_2 = n'_2*$ for some n'_1 and n'_2 : It follows from (31), (46), (149), and (146) that

$$\begin{aligned} S^{\mathrm{z}}_0 &\rightarrow_0 S^{\mathrm{z}}_1 && (150) \\ S^{\mathrm{z}}_1 &= (M^{\mathrm{z}}_0; e^{\mathrm{z}}_1) && (151) \\ e^{\mathrm{z}}_0 &= \langle\ n* \triangleright \mathrm{r}, && (152) \\ &\quad n'* \triangleright \mathrm{g}, \\ &\quad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (n_2* \triangleright \mathrm{b})\ \rangle \\ n' &= n'_1 + n'_2 && (153) \end{aligned}$$

Applying (31) and (46) again, we get

$$\begin{aligned} S^{\mathrm{z}}_1 &\rightarrow_0 S^{\mathrm{z}}_2 && (154) \\ S^{\mathrm{z}}_2 &= (M^{\mathrm{z}}_0; e^{\mathrm{z}}_2) && (155) \\ e^{\mathrm{z}}_2 &= \langle\!\langle n*, n'*, n* \rangle\!\rangle && (156) \end{aligned}$$

From (147), (156), (155), (137), (154), (150), and (48), there exists $S^{\mathrm{z}} = S^{\mathrm{z}}_2$ such that $S_0 \rightarrow^*_0 S^{\mathrm{z}}$ and $\llbracket S' \rrbracket\varepsilon \succ_c S^{\mathrm{z}}$, as required.

Case $v_1 \neq n'_1*$ or $v_2 \neq n'_2*$ for some n'_1 and n'_2 : It follows from (32) and from (46) and from (149) and (146) that

$$\begin{aligned} S^{\mathrm{z}}_0 &\rightarrow_0 S^{\mathrm{z}}_1 && (157) \\ S^{\mathrm{z}}_1 &= (M^{\mathrm{z}}_0; e^{\mathrm{z}}_1) && (158) \\ e^{\mathrm{z}}_1 &= \langle\ n*, && (159) \\ &\quad v, \\ &\quad (n_1* \triangleright \mathrm{b}) +_{\mathrm{b}} (n_2* \triangleright \mathrm{b})\ \rangle \end{aligned}$$

for some v. Applying (31) and (46) using (159) and (158) yields

$$\begin{aligned} S^{\mathrm{z}}_1 &\rightarrow_0 S^{\mathrm{z}}_2 && (160) \\ S^{\mathrm{z}}_2 &= (M^{\mathrm{z}}_0; e^{\mathrm{z}}_2) && (161) \\ e^{\mathrm{z}}_2 &= \langle\!\langle n*, v, n* \rangle\!\rangle && (162) \end{aligned}$$

Now, from (147), (162), (161), (137), (160), (157), and (48), there exists $S^{\mathrm{z}} = S^{\mathrm{z}}_2$ such that $S_0 \rightarrow^*_0 S^{\mathrm{z}}$ and $\llbracket S' \rrbracket\varepsilon \succ_c S^{\mathrm{z}}$, as required.

$\boxed{\text{CASE } S = (M; E[e_{\mathrm{s}}]);\ \text{rule (46)}}$ From assumptions, we have

$$\begin{aligned} S_{\mathrm{s}} &\hookrightarrow S'_{\mathrm{s}} && (163) \\ S &= E[S_{\mathrm{s}}] && (164) \\ S' &= E[S'_{\mathrm{s}}] && (165) \end{aligned}$$

where $S' = (M'; E[e'_{\mathrm{s}}])$ and $S_{\mathrm{s}} = (M; e_{\mathrm{s}})$ and $S'_{\mathrm{s}} = (M; e'_{\mathrm{s}})$. From assumptions, (164), (165), and Proposition 19, we have there exists E' such that

$$\begin{aligned} E'[\llbracket S_{\mathrm{s}} \rrbracket\varepsilon] = \llbracket E[S_{\mathrm{s}}] \rrbracket\varepsilon = \llbracket S \rrbracket\varepsilon && (166) \\ E'[\llbracket S'_{\mathrm{s}} \rrbracket\varepsilon] = \llbracket E[S'_{\mathrm{s}}] \rrbracket\varepsilon = \llbracket S' \rrbracket\varepsilon && (167) \end{aligned}$$

Now, because $\llbracket S_{\mathrm{s}} \rrbracket\varepsilon$ and $\llbracket S'_{\mathrm{s}} \rrbracket\varepsilon$ exist, we can apply Proposition 14, using (163), to get

$$\llbracket S_{\mathrm{s}} \rrbracket\varepsilon \rightarrow^*_0 \llbracket S'_{\mathrm{s}} \rrbracket\varepsilon \tag{168}$$

From (166), (167), and assumptions, we have

$$E'[\llbracket S_{\mathrm{s}} \rrbracket\varepsilon] \rightarrow^*_0 S_0 \rightarrow^*_0 E'[\llbracket S'_{\mathrm{s}} \rrbracket\varepsilon] \tag{169}$$

From (168) and repeated use of Proposition 11, we have

$$\begin{aligned} E'[\llbracket S_{\mathrm{s}} \rrbracket\varepsilon] \rightarrow_0 E'[S_1] \rightarrow_0 \ldots \rightarrow_0 E'[S_n] \rightarrow_0 E'[\llbracket S'_{\mathrm{s}} \rrbracket\varepsilon] && (170) \\ \llbracket S_{\mathrm{s}} \rrbracket\varepsilon \rightarrow_0 S_1 \rightarrow_0 \ldots \rightarrow_0 S_n \rightarrow_0 \llbracket S'_{\mathrm{s}} \rrbracket\varepsilon && (171) \end{aligned}$$

Now, because evaluation is deterministic (Proposition 9) and by comparing (169) and (170), we have, using (171), there exists S'_0 such that

$$\begin{aligned} S_0 &= E'[S'_0] && (172) \\ \llbracket S_{\mathrm{s}} \rrbracket\varepsilon \rightarrow^*_0 S'_0 &\rightarrow^*_0 \llbracket S'_{\mathrm{s}} \rrbracket\varepsilon && (173) \end{aligned}$$

Now, let S^{z}_0 be any machine state and c any color r, g, or b such that

$$S_0 \succ_c S^{\mathrm{z}}_0 \tag{174}$$

From (174), (172), and Proposition 18, we have there exist E'' and S^{u}_0 such that

$$\begin{aligned} S^{\mathrm{z}}_0 &= E''[S^{\mathrm{u}}_0] && (175) \\ S'_0 &\succ_c S^{\mathrm{u}}_0 && (176) \\ E' &\succ_c E'' && (177) \end{aligned}$$

From (163) and (173), it follows by induction, using (176), that there exists S^{u} such that

$$\begin{aligned} S^{\mathrm{u}}_0 &\rightarrow^*_0 S^{\mathrm{u}} && (178) \\ \llbracket S'_{\mathrm{s}} \rrbracket\varepsilon &\succ_c S^{\mathrm{u}} && (179) \end{aligned}$$

Let

$$S^{\mathrm{z}} = E''[S^{\mathrm{u}}] \tag{180}$$

From Proposition 11, (178), (175), and (180), we have $S_0^{\mathrm{z}} \to_0^* S^{\mathrm{z}}$, as required. Moreover, from (177), (167), (179), (180), and Proposition 17, we also have $[\![S']\!]\varepsilon \succ_c S^{\mathrm{z}}$, as required. □

4.4 Multi-Step Fault Tolerance

We can now use the fault-tolerance proposition from Section 4.3 to yield the following "multi-step" fault-tolerance property:

PROPOSITION 22 (Multi-Step Fault-Tolerance). *Assume $S \hookrightarrow^* S'$ and $[\![S]\!]\varepsilon \to_0^* S_0 \to_0^* [\![S']\!]\varepsilon$.*

For any S_0^{z} such that $S_0 \succ S_0^{\mathrm{z}}$ there exists S^{z} such that $S_0^{\mathrm{z}} \to_0^* S^{\mathrm{z}}$ and $[\![S']\!]\varepsilon \succ S^{\mathrm{z}}$.

PROOF By induction on the length of the derivation $S \hookrightarrow^* S'$, using Proposition 21. □

Intuitively, this proposition is obtained by composing diagrams of the form shown in Section 4.3.

4.5 Putting it All Together

An important feature of the fault propagation relation is that it supports voting when used together with the faulty translation. We define *voting* as a binary relation $\approx_{\mathrm{vote}}$ between source language terminal machine states and target language terminal machine states.

$$\begin{array}{lcll} (M;v) & \approx_{\mathrm{vote}} & (M';\langle\!\langle v',v,v\rangle\!\rangle) & \text{if}[\![M]\!] \succ_{\mathrm{r}} M' \\ (M;v) & \approx_{\mathrm{vote}} & (M';\langle\!\langle v,v',v\rangle\!\rangle) & \text{if}[\![M]\!] \succ_{\mathrm{g}} M' \\ (M;v) & \approx_{\mathrm{vote}} & (M';\langle\!\langle v,v,v'\rangle\!\rangle) & \text{if}[\![M]\!] \succ_{\mathrm{b}} M' \end{array}$$

The following property holds:

PROPOSITION 23 (Fault Propagation Supports Voting). *If $[\![S_{\mathrm{v}}]\!]\varepsilon \succ S_{\mathrm{v}}^{\mathrm{z}}$ then $S_{\mathrm{v}} \approx_{\mathrm{vote}} S_{\mathrm{v}}^{\mathrm{z}}$*

PROOF Follows immediately from the definition of translation of values. □

We can now tie all the previous properties together in the following top-level fault-tolerance property:

THEOREM 1 (Top-level Fault Tolerance). *Assume $\vdash^{\mathrm{s}} S : \tau$ and $k \leq 1$.*

1. *Either there exists S_{v} such that $S \hookrightarrow^* S_{\mathrm{v}}$ in which case there exists $S_{\mathrm{v}}^{\mathrm{z}}$ such that $[\![S]\!]\varepsilon \to_k^* S_{\mathrm{v}}^{\mathrm{z}}$ and for all such $S_{\mathrm{v}}^{\mathrm{z}}$ it holds that $S_{\mathrm{v}} \approx_{\mathrm{vote}} S_{\mathrm{v}}^{\mathrm{z}}$.*
2. *Otherwise $S\!\uparrow$ in which case $[\![S]\!]\varepsilon\!\uparrow_k$.*

PROOF From Proposition 4, it follows that either there exists S_{v} such that $S \hookrightarrow^* S_{\mathrm{v}}$ or $S\!\uparrow$.

We now consider part one of the proposition, with $k = 1$. From assumptions and from Proposition 10, we have $[\![S]\!]\varepsilon$ exists. From assumptions and repeated use of Proposition 6, we have $\vdash^{\mathrm{s}} S_{\mathrm{v}} : \tau$, thus, from Proposition 10, we have $[\![S']\!]\varepsilon$ exists. Now, let $S_{\mathrm{v}}^{\mathrm{z}}$ be any terminal machine state such that

$$[\![S]\!]\varepsilon \quad \to_k^* \quad S_{\mathrm{v}}^{\mathrm{z}} \tag{181}$$

From (181) and due to the property that non-faulty evaluation is deterministic (Proposition 9), there exist S_0 and S_0^{z} such that

$$\begin{array}{rclcll} [\![S]\!]\varepsilon & \to_0^* & S_0 & \to_0^* & [\![S_{\mathrm{v}}]\!]\varepsilon & (182) \\ & & S_0 & \to_1 & S_0^{\mathrm{z}} & (183) \\ & & S_0^{\mathrm{z}} & \to_0^* & S_{\mathrm{v}}^{\mathrm{z}} & \end{array}$$

From (183) and Proposition 20, we have

$$S_0 \quad \succ \quad S_0^{\mathrm{z}} \tag{184}$$

Now, from assumptions, (182), (184), and Proposition 22, there exists S^{z} such that

$$[\![S_{\mathrm{v}}]\!]\varepsilon \quad \succ \quad S_{\mathrm{v}}^{\mathrm{z}} \tag{185}$$

From (185) and Proposition 23, we have $S_{\mathrm{v}} \approx_{\mathrm{vote}} S_{\mathrm{v}}^{\mathrm{z}}$, as required.

For $k = 0$, part one of the proposition is similar but simpler than the case for $k = 1$.

For $k = 0$, part two of the proposition follows immediately from Proposition 14. For $k = 1$, part two follows from Proposition 14 and from Proposition 21. □

Notice that in the proof of Proposition 1, we make essential use of the property that non-faulty evaluation is deterministic (Proposition 9). It is thus important that machine states are considered identical up-to consistent renaming of labels.

5. Conclusion and Future Work

We have presented a simply typed lambda calculus extended with two simple atomic operations that allows the calculus to be the target of a fault-tolerant translation that allows single-transient faults to appear at runtime, without effecting the overall result of evaluation.

There are plenty of possibilities for future work. First, the scheme does not support *full fault recovery* in the sense that after a fault, the system can recover to a state where it is ready for a new arbitrary fault. This problem is also present in previous work [18]. A weaker *partial recovery* property may be easier to establish. Let two function invocations be *unrelated* if none of the function invocations makes a function call that directly results in the other function invocation. A partial fault recovery property may state that as long as two transient faults occur during execution of unrelated function invocations then execution can proceed safely. Such a partial fault recovery property may be implemented by majority voting on function arguments and results and on storing closure environments (i.e., on substitutions under lambda-bindings in our formulation).

Second, another interesting direction for future work would be to investigate techniques for supporting fault-tolerant control registers, perhaps through bootstrapping [3] a faulty-translated interpreter written in an appropriately extended version of the source language. Similarly, by writing a garbage collector in such a source language, the faulty translation may result in a fault-tolerant garbage collector for use by the system.

Third, another interesting direction for future work is to investigate type systems for capturing the soundness of the embedded voting instructions, which could be a first step towards a lower-level type safe language for full transient-fault tolerant computing [8].

Also identified in [18] as possible future work is to investigate the possibilities for optimizations of generated fault-tolerant code. Whereas simple in-lining, constant propagation, and constant folding appear to be sound optimizations, common subexpression elimination across colors is not sound in general.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. November 2005. Alexandria, VA.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. November 2005. Manchester, U.K.

[3] Ravi Sethi Alfred V. Aho and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison Wesley, 1986.

[4] Robert C. Baumann. Soft errors in advanced semiconductor devices—part I: The three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, March 2001.

[5] Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 83–92, 2006.

[6] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. August 2006. Vancouver, BC, Canada.

[7] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1995.

[8] Frances Perry, Lester Mackey, George Reis, Jay Ligatti, David August, and David Walker. Fault-tolerant typed assembly language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2007. San Diego, California, USA.

[9] M. Rebaudengo, M. Reorda, M. Torchiano, and M. Violante. A source-to-source compiler for generating dependable software. In *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

[10] M. Rebaudengo, M. Reorda, and M. Violante. A new software-based technique for low-cost fault-tolerant application. In *IEEE Reliability and Maintainability Symposium*, pages 25–28, 2003.

[11] George A. Reis, David I. August, Robert Cohn, and Shubhendu S. Mukherjee. Software fault detection using dynamic instrumentation. In *Proceedings of the Fourth Annual Boston Area Architecture Workshop (BARC)*, February 2006.

[12] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[13] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, 2(4):366–396, 2005.

[14] Goutam Kumar Saha. Software based fault tolerance—a survey. *Ubiquity*, 7(25), July 2006. www.acm.org/ubiquity.

[15] Úlfar Erlingsson, Martn Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. November 2006. Seattle, Washington.

[16] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. July 1999. Revised version.

[17] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[18] David Walker, Lester Mackey, Jay Ligatti, George Reis, and David August. Static typing for a faulty lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2006.

[19] Nobuko Yoshida, Kohei Honda, and Martin Berger. Logical reasoning for higher-order functions with local state. pages 361–377, March 2007. Braga, Portugal.