Verifying Compiler Transformations for Concurrent Programs

March 11, 2009

Technical Report
MSR-TR-2008-171

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Verifying Compiler Transformations for Concurrent Programs

Sebastian Burckhardt Madanlal Musuvathi
Microsoft Research
{sburckha,madanm}@microsoft.com

Vasu Singh
EPFL, Switzerland
vasu.singh@epfl.ch

Abstract

Compilers transform programs, either to optimize performance or to translate language-level constructs into hardware primitives. For concurrent programs, ensuring that a transformation preserves the semantics of the input program can be challenging. In particular, the emitted code must correctly emulate the semantics of the language-level memory model when running on hardware with a relaxed memory model.

In this paper, we present a novel proof methodology for proving the soundness of compiler transformations for concurrent programs. Our methodology is based on a new formalization of memory models as dynamic rewrite rules on event streams. We implement our proof methodology in a first-of-its-kind semi-automated tool called Traver to verify or falsify compiler transformations. Using Traver, we prove or refute the soundness of several commonly used compiler transformations for various memory models. In this process, we find subtle bugs in the CLR JIT compiler and in the JSR-133 Java JIT compiler recommendations.

Keywords Compiler correctness, concurrency, memory models

1. Introduction

The correctness of compilers is crucial for the reliability of modern software systems. Compilers perform a series of transformations that translate a high-level program into low-level machine instructions, while optimizing the code for performance. For correctness, these transformations have to preserve the meaning for any input program. Proving the correctness of these transformations has been extensively studied for sequential programs [19, 13, 11, 14]. In this paper, we focus on the correctness of compiler transformations for multithreaded programs.

The presence of concurrency (not surprisingly) substantially complicates the reasoning when trying to establish correctness of transformations. First, the effects of a transformation are no longer local. Two program snippets that are functionally equivalent with respect to the input-output behavior can still exhibit different interactions with other threads in the system. Thus, a seemingly correct transformation can introduce subtle safety or liveness errors in an otherwise correct program. Such errors are extremely difficult to find after the fact, because they may require specific user programs, thread interleavings, and hardware multiprocessor configurations to manifest at runtime.

Second, the semantics of a concurrent program is defined by a language memory model [15, 2]. In order to facilitate certain optimizations, these memory models allow more behaviors than sequential consistency, the most intuitive model for programmers. Obviously, any proof methodology for the correctness of compiler transformations should account for these relaxations. In particular, the effect of each transformation should be in accordance with the officially specified memory model. Manually doing so can be extremely error-prone [18].

Finally, the correctness of transformations depends also on the memory model of the underlying hardware. Machine code can be further transformed dynamically in the hardware as it executes. Compilers should ensure that the generated code retains the semantics of the original program in the presence of such relaxations. This is particularly important for backend compiler transformations that map language constructs to hardware primitives and for those that perform optimizations on the machine code. Again, any proof methodology should account for these hardware memory models as well. Section 2 contains examples that elucidate all of the challenges mentioned above.

To address these challenges, we present a proof methodology and a tool for verifying program transformations for concurrent programs. We first establish a formal definition (§3.3) for the soundness of program transformations. Our definition is based on a general notion of "observations" of an execution that hides internal program details but preserves *all* externally visible events, including program termination.

We then introduce a new memory model formalization (§3.4) that unifies the specification of both language and hardware memory models as rewrite rules on program events. This new formalization allows mechanical reasoning of the effects of memory models on program transformations. At the same time, our formalization is powerful enough to concisely state realistic memory models. In this paper, we provide a formal specification of the x86 memory model [10] that is precise enough to capture the non-atomic behavior of hardware stores (§3.4.2). We also provide, to our knowledge, the first formal specification of the working CLR memory model [3, 8, 16] (§3.4.3).

Finally, we present a semi-automated tool called Traver (short for transformation verifier) that implements a novel proof methodology for verifying or falsifying the soundness of program transformations against memory models. Given a local program transformation and a memory model, Traver uses an automated theorem prover [7] to prove that the set of observations of the transformed program is contained in the set of observations of the original program, for all possible program contexts. Conversely, when provided with an additional falsification context, Traver can automatically show that the transformation produces an outcome that is observable. This produces a certificate of unsoundness of the transformation.

Using Traver, we determine the soundness of several commonly used compiler transformations for different memory models (§6). In this process, Traver discovered two subtle but important bugs (§6) that refute accepted-wisdom among compiler experts. The first bug is in the CLR JIT compiler for the x86 platform. Due to the store-load forwarding in x86, we show that volatile reads in C# are not guaranteed to have acquire semantics. The second bug is in the recommendations of the popular cookbook [12] for JVMs. We show that, despite strong ordering guarantees in x86, a fence between two volatile loads cannot be eliminated.

```
bool B = false, X = false, Y = false;

Transformation                          Observer

bool r = B;
if(r) {
  X = r; Y = !r;          bool r = B;
} else {            ⇒     X = r;        X = true;
  Y = !r; X = r;          Y = !r;       assert(X || Y);
}
```

Figure 1. An incorrect transformation that reorders writes to shared variables `X` and `Y`.

```
int X;

int r1 = X;          int r1 = X;
int r2 = X;    ⇒     int r2 = r1;
```

Figure 2. Redundant read elimination

Our experience with Traver has successfully demonstrated the power of formalism and automation in discovering corner cases where normal programmer intuition fails. However, we believe that the proof methodology and the tool presented in the paper has many more uses in the future. First, Traver can be integrated in an existing compiler to automatically verify compiler transformations. Second, current compiler optimizations for concurrent programs are fairly conservative. A tool like Traver opens the possibility for new, aggressive optimizations for concurrent programs that can then be proven correct. Finally, we believe that Traver is also useful for designing memory models for new concurrent programming languages, as it allows the language designer to explore various tradeoffs between performance and expressiveness of the language.

2. Motivating Examples

In this section, we provide examples to motivate our work and to introduce the readers to the challenges involved in proving compiler transformations for concurrent programs.

Figure 1 shows a program transformation that is correct only for sequential programs. This program[1] contains three shared variables (capitalized by convention) and a local variable `r`. The transformation removes a branch whose outcomes have the same net effect. However, the transformation requires a reordering of shared-variable stores in the else-branch. This can be observed by the thread on the right in Figure 1. To see this, the assert cannot fail in the original program, as the only branch outcome (the else branch) that sets `X` to false sets `Y` to true before the update to `X`. The transformed program does not provide this guarantee, potentially causing an assertion violation.

Figure 2 shows a correct transformation that contains the elimination of a redundant read of the variable `X`. The correctness, however, is not entirely obvious. One has to show that this program does not change the behavior of every conceivable "observer" thread. Using the techniques described in this paper, one can indeed prove that this is so.

Readers should note that this transformation *reduces* the set of behaviors of the program. the local variables `r1` and `r2` can be different in the original program, if the computation is interleaved with other threads that modify `X`, but not so in the modified program. We deem it acceptable for program transformations to reduce the set of behaviors resulting from the nondeterminism of the scheduler. Many useful transformations would not be possible without this flexibility.

[1] Assume sequential consistency for now.

```
int X = 0;

Transformation                          Observer

int r1 = X;                 int r1 = X;
while(X == 0);        ⇒     while(r1 == 0);     X = 1;
```

Figure 3. Redundant read elimination causing nontermination.

```
int X = 0, A = 1, B = 1

Transformation                          Observer

int r = X;
r = r + A;          int r = X;          B = 0;
X = r;              r = r + A;          mfence;
r = X;        ⇒     r = r + B;          if(X == 0)
r = r + B;          X = r;                assert(X <= 1);
X = r;
```

Figure 4. The transformation above is not correct for sequentially consistency, but is correct for the x86 memory model.

```
volatile int X=0,Y=0;

Transformation                          Observer

// C# program       // x86 program
int r1,r2;          register r1,r2;     int r3;

                    ⇒
X = 2;              mov X,2;            Y = 1;
r1 = X;             mov r1,X;           X = 1;
r2 = Y;             mov r2,Y;           r3 = X;
```

Figure 5. CLR-JIT bug: The store-load forwarding of `X` to `r1` in x86 breaks the acquire-semantics required by the CLR memory model. The final state `X=2,Y=1,r1=2,r2=0,r3=2` is not valid for the C# program, but is possible in the translated x86 program.

On the other hand, a transformation is not allowed to *arbitrarily* reduce behaviors — it is still important to retain behaviors that ensure the *liveness* of programs. Figure 3 shows a transformation very similar to the one in Figure 2 but is not correct. With the observer in the figure, the original program terminates but the transformed program does not. Our proof methodology can identify transformations that introduce such liveness violations.

Figure 4 shows a transformation that is correct only for certain memory models. The original program performs two increments of `X`, first by `A` and then by `B`, writing back the value of `X` after each increment. The transformation optimizes the intermediate store of `X`. The observer thread in Figure 4 shows that this transformation is not correct for sequential consistency. With the original program, the branch in the observer is taken when the program has not updated `X` yet. But by this time, the observer has set `B` to zero, ensuring that the assert never fails. This is not the case with the transformed program. This transformation is (surprisingly) correct for the x86 memory model. Under the x86 memory model, regular stores can be delayed past subsequent loads due to the presence of store-buffers in the hardware. Thus, the assertion in the observer can fire both before and after the transformation. The proof methodology can be used to show that no observer can differentiate the two programs in Figure 4 under the x86 memory model.

Figure 5 describes a bug in the CLR JIT that our tool found (§6). This demonstrates the difficulting of retaining the language mem-

ory model in the presence of the hardware memory model. The working CLR memory model [3, 8, 16], unlike its Java counterpart [15], does not guarantee sequential consistency for volatile accesses. However, it provides acquire semantics to volatile reads and release semantics to volatile writes. In particular, a volatile store can be reordered with a subsequent volatile read, but volatile store-store and volatile load-load reorderings are not allowed. As the x86 provides similar ordering guarantees, the CLR-x86 JIT translates volatile accesses into regular x86 accesses. Contrary to expert-intuition, this is wrong.

We informally describe the bug below. Section 6 contains a formal description. With the observer in Figure 5, a final state of `X=2,Y=1,r1=2,r2=0,r3=2` is not reachable in the CLR memory model. The final values of `r1` and `r3` restrict the store-load reorderings allowed by the CLR memory model, requiring sequential consistency for this code fragment. The reader should convince herself that under this restriction `r2=0` is not possible. The x86 memory model however, allows "store-load-forwarding". This means that the store to `X` can be delayed past the load of `Y` while `r1` still receiving the forwarded value from this store.

3. Formulation

In our experience, concurrency and relaxed memory models often go beyond what our intuition can handle. We now establish a formalism for programs, program transformations, and memory models that lets us precisely state, prove, and refute the soundness of transformations.

3.1 The calculus

We start with a simple imperative calculus for shared-memory concurrent programs. We define the syntax of *snippets* (program fragments) recursively, in Fig. 6.

Our calculus distinguishes between shared variables $\mathcal{L}$ (which may be accessed concurrently) and local variables[2] $\mathcal{R}$. We let $\mathcal{V} = \mathcal{L} \cup \mathcal{R}$ be the set of all variables. We let $\mathcal{X}$ be the set of values assumed by the variables. For simplicity, we only use integers here, assuming $\mathcal{X} = \mathbb{Z}$.

The (load) and (store) statements move values between local and shared variables; we use an access qualifier $h \in \mathcal{H}$ (defined in Section 3.4) to distinguish different access types, such as normal versus volatile. The (assign) statement performs computation, such as addition, on local variables. The (compare-and-swap) statement[3] compares the values of L and r_c, stores r_n to L if they are equal, and assigns the original value of L to r_r.

The statements (get) and (print) represent simple I/O in the form of reading from or writing to an interactive console. The statements (sequential composition), (conditional) and (loop) have their usual meaning (we let 0 denote false, and all other numbers denote true). The statement (parallel composition) executes its components concurrently, and waits for all of them to finish before completing. The statements (local) and (shared) declare *mutable* variables and initialize them to the given value. [4]

To enforce that local variables are not accessed concurrently, we define the free variables as in (Fig. 7) and call a snippet *ill-typed* if it contains a parallel composition $s_1 \parallel \cdots \parallel s_n$ such that for some i, j, we have $(FV(s_i) \cap FV(s_j) \cap \mathcal{R}) \neq \emptyset$, and *well-typed* otherwise. We let $\mathcal{S}$ be the set of all well-typed snippets.

Finally, we define a *program* to be a well-typed snippet s with no free variables. We let $\mathcal{P}$ be the set of all programs.

L	$\in$	$\mathcal{L}$	(shared variable)
r	$\in$	$\mathcal{R}$	(local variable)
x	$\in$	$\mathcal{X}$	(value)
h	$\in$	$\mathcal{H}$	(access qualifier)
f	:	$\mathcal{X}^n \to \mathcal{X}$	(local computation), $n \geq 0$
s	$::=$	$\mathbf{skip}$	(skip)
	$\mid$	$r :=_h L$	(load)
	$\mid$	$L :=_h r$	(store)
	$\mid$	$r := f(r_1, \ldots, r_n)$	(assign), $n \geq 0$
	$\mid$	$r_r := cas_h(L, r_c, r_n)$	(compare and swap)
	$\mid$	$get\ r$	(read from console)
	$\mid$	$print\ r$	(write to console)
	$\mid$	$s;\ s$	(sequential composition)
	$\mid$	$s_1 \parallel \cdots \parallel s_n$	(parallel composition), $n \geq 2$
	$\mid$	$\mathbf{if}\ r\ \mathbf{then}\ s\ \mathbf{else}\ s$	(conditional)
	$\mid$	$\mathbf{while}\ r\ \mathbf{do}\ s$	(loop)
	$\mid$	$\mathbf{local}\ r = x\ \mathbf{in}\ s$	(local variable declaration)
	$\mid$	$\mathbf{share}\ L = x\ \mathbf{in}\ s$	(shared variable declaration)

Figure 6. Syntax of program snippets s.

$$
\begin{aligned}
FV(\mathbf{skip}) &= \emptyset \\
FV(r :=_h L) &= \{r, L\} \\
FV(L :=_h r) &= \{L, r\} \\
FV(r_0 := f(r_1 \ldots r_n)) &= \{r_0, r_1, \ldots r_n\} \\
FV(r_r := cas_h(L, r_c, r_n)) &= \{L, r_r, r_c, r_n\} \\
FV(get\ r) &= \{r\} \\
FV(print\ r) &= \{r\} \\
FV(s; s') &= FV(s) \cup FV(s') \\
FV(s_1 \parallel \cdots \parallel s_n) &= FV(s_1) \cup \cdots \cup FV(s_n) \\
FV(\mathbf{if}\ r\ \mathbf{then}\ s\ \mathbf{else}\ s') &= \{r\} \cup FV(s) \cup FV(s') \\
FV(\mathbf{while}\ r\ \mathbf{do}\ s) &= \{r\} \cup FV(s) \\
FV(\mathbf{local}\ r = x\ \mathbf{in}\ s) &= FV(s) \setminus \{r\} \\
FV(\mathbf{share}\ L = x\ \mathbf{in}\ s) &= FV(s) \setminus \{L\}
\end{aligned}
$$

Figure 7. Definition of the set of free variables $FV(s)$ of s.

3.2 Observations

In order to arrive at a comprehensive definition of soundness for program transformations, we distinguish between externally observable effects and internal details. A sound transformation should not change externally observable effects of a program. We let these effects include (1) whether the program terminates or diverges, and (2) the sequence of visible events the program produces when interacting with its environment.

Formally, let Ext be the set of of externally visible events. In real programs, there can be many such visible events due to the wealth of interactions with the environment. In our calculus, these include reading a number from an interactive console, and printing a number to the console:

$$Ext = \{\langle get\ n \rangle \mid n \in \mathbb{Z}\} \cup \{\langle print\ n \rangle \mid n \in \mathbb{Z}\}$$

Let $Ext^\infty = Ext^* \cup Ext^\omega$ be the set of finite or infinite sequences of events, and let ϵ denote the empty sequence. Now we define the set of *observations* as follows:

$$\mathcal{O} = \{u \mid u \in Ext^*\} \cup \{\nabla u \mid u \in Ext^\infty\}$$

[2] Local variables are intended to represent non-shared entities like registers or non-escaping local variables, while shared variables represent all memory locations for which the compiler can not prove that they are thread-local.

[3] Compare-and-swap provides a simple, yet universal primitive from which other synchronization operations can be derived [9].

[4] Compared to *let*, as used in functional languages, they differ by (1) allowing mutation of the variable, and (2) strictly restricting the scope and lifetime to the nested snippet.

$$p_1 = \left[\begin{array}{l} \textbf{local } r = 1 \textbf{ in} \\ \textbf{local } s = 2 \textbf{ in} \\ (\mathit{print}\ r)\ \|\ (\mathit{print}\ s) \end{array}\right] \quad p_2 = \left[\begin{array}{l} \textbf{local } r = 1 \textbf{ in} \\ \textbf{local } s = 2 \textbf{ in} \\ \mathit{print}\ r; \\ \mathit{print}\ s \end{array}\right]$$

$$p_3 = \left[\begin{array}{l} \textbf{local } r = 1 \textbf{ in} \\ \textbf{while } r \textbf{ do} \\ \quad \mathit{print}\ r \end{array}\right] \quad p_4 = \left[\begin{array}{l} \textbf{local } r = 0 \textbf{ in} \\ \mathit{get}\ r; \\ \textbf{while } r \textbf{ do} \\ \quad \textbf{skip}; \\ \mathit{print}\ r \end{array}\right]$$

Figure 8. Four example programs. p_1 and p_2 always terminate, p_3 never terminates, and p_4 sometimes terminates. p_1 can be soundly transformed to p_2, but not vice versa.

$$\begin{array}{rl} c ::= & [\,] \quad | \quad c\,;\,s \quad | \quad s\,;\,c \quad | \quad \textbf{while } r \textbf{ do } c \\ | & \textbf{if } r \textbf{ then } c \textbf{ else } s \quad | \quad \textbf{if } r \textbf{ then } s \textbf{ else } c \\ | & s_1 \,\|\, \cdots \,\|\, s_{k-1} \,\|\, c \,\|\, s_{k+1} \,\|\, \cdots \,\|\, s_n \quad (1 \le k \le n) \\ | & \textbf{local } r = x \textbf{ in } c \quad | \quad \textbf{share } L = x \textbf{ in } c \end{array}$$

Figure 9. Definition of the syntax of program contexts.

An observation of the form u represents a terminating execution that produces the finite event sequence u; an observation of the form ∇u represents a nonterminating execution that produces the (finite or infinite) sequence u. For example, the program p_1 in Fig. 8 has two possible observations, $\langle \mathit{print}\ 1\rangle\langle \mathit{print}\ 2\rangle$ and $\langle \mathit{print}\ 2\rangle\langle \mathit{print}\ 1\rangle$; the program p_2 has one possible observation, $\langle \mathit{print}\ 1\rangle\langle \mathit{print}\ 2\rangle$; the program p_3 has one possible observation, $\nabla\langle \mathit{print}\ 1\rangle^\omega$; and the program p_4 has the following set of possible observations:

$$\{\langle \mathit{get}\ 0\rangle\langle \mathit{print}\ 0\rangle\} \cup \{\nabla\langle \mathit{get}\ n\rangle \mid n \neq 0\}$$

3.3 Transformations

Having defined exactly what can be observed about program executions, we can define soundness of transformations based on the idea that their effects should not be observable. For programs $p, p' \in \mathcal{P}$, we let $p \Rightarrow p'$ represent the *program transformation* of p into p'.

DEFINITION 1. *We call a program transformation $p \Rightarrow p'$* sound *if $obs(p') \subseteq obs(p)$.*

An important point is that we consider it acceptable if the transformed program has *fewer* observations than the original one. For example, we would consider it o.k. to transform program p_1 to program p_2 in Fig. 8, which essentially reduces the nondeterministic choices available to the scheduler in scheduling the two print statements. An external entity interacting with the program cannot observe this reduction.

Of particular practical relevance are local transformations. We let a *program context* be a "program with a hole []", defined in Fig. 9. For a context c and snippet s, we let $c[s]$ be the snippet obtained by replacing the hole in c with s. For two snippets $s, s' \in \mathcal{S}$, we let $(s \rightarrow s')$ be a *transformation rule*.

DEFINITION 2. *We say a transformation rule $(s \rightarrow s')$* induces *a program transformation $p \Rightarrow p'$ if there exists a context c such that $p = c[s]$, $p' = c[s']$. We call a local transformation rule $(s \rightarrow s')$* sound *if all induced program transformations $p \Rightarrow p'$ are sound.*

$$\begin{array}{rcll} L & \in & \mathcal{L} & \text{(shared variable)} \\ x & \in & \mathcal{X} & \text{(value)} \\ h & ::= & R \mid L \mid W \mid S & \text{(access qualifier)} \\ e & ::= & \langle ld_h\ L, x\rangle & \text{(load)} \\ & | & \langle st_h\ L, x\rangle & \text{(store)} \\ & | & \langle ldst_h\ L, x_l, x_s\rangle & \text{(atomic load-store)} \\ & | & \langle \mathit{mfence}\rangle & \text{(full memory fence)} \\ & | & \langle \mathit{get}\ x\rangle & \text{(get)} \\ & | & \langle \mathit{print}\ x\rangle & \text{(print)} \end{array}$$

Figure 10. Definition of events e.

Sequential Consistency (SC)

$e ::= \langle ld_S\ L, x\rangle \mid \langle st_S\ L, x\rangle \mid \langle ldst_S\ L, x_l, x_s\rangle$

(no rewrite rules)

Intel 32/64 Model $(X86)$

$e ::= \langle ld_R\ L, x\rangle \mid \langle st_R\ L, x\rangle \mid \langle \mathit{mfence}\rangle$
$\quad \mid \langle ld_L\ L, x\rangle \mid \langle st_L\ L, x\rangle \mid \langle ldst_L\ L, x_l, x_s\rangle$

srsrl (swap regular store - regular load) $=$
$\langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle \overset{L \neq L'}{\rightarrow} \langle ld_R\ L', x'\rangle\langle st_R\ L, x\rangle$

ersrlf (eliminate regular store - regular load - forward) $=$
$\langle st_R\ L, x\rangle\langle ld_R\ L, x\rangle \rightarrow \langle st_R\ L, x\rangle$

Informal CLR 2.0 Model (CLR_{blog})

$e ::= \langle ld_W\ L, x\rangle \mid \langle st_W\ L, x\rangle$
$\quad \mid \langle ld_S\ L, x\rangle \mid \langle st_S\ L, x\rangle \mid \langle ldst_S\ L, x_l, x_s\rangle$

ssl (swap store - load) $=$
$\langle st_h\ L, x\rangle\langle ld_{h'}\ L', x'\rangle \overset{L \neq L'}{\rightarrow} \langle ld_{h'}\ L', x'\rangle\langle st_h\ L, x\rangle$

swll (swap weak load - load) $=$
$\langle ld_W\ L, x\rangle\langle ld_h\ L', x'\rangle \rightarrow \langle ld_h\ L', x'\rangle\langle ld_W\ L, x\rangle$

eswlf (eliminate store - weak load - forward) $=$
$\langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle \rightarrow \langle st_h\ L, x\rangle$

Informal CLR Model with SC volatiles (CLR^{SC}_{blog})

$e ::= \langle ld_W\ L, x\rangle \mid \langle st_W\ L, x\rangle$
$\quad \mid \langle ld_S\ L, x\rangle \mid \langle st_S\ L, x\rangle \mid \langle ldst_S\ L, x_l, x_s\rangle$

swswl (swap weak store - weak load) $=$
$\langle st_W\ L, x\rangle\langle ld_W\ L', x'\rangle \overset{L \neq L'}{\rightarrow} \langle ld_W\ L', x'\rangle\langle st_W\ L, x\rangle$

swlwl (swap weak load - weak load) $=$
$\langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle \rightarrow \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle$

ewswlf (eliminate weak store - weak load - forward) $=$
$\langle st_W\ L, x\rangle\langle ld_W\ L, x\rangle \rightarrow \langle st_W\ L, x\rangle$

Figure 11. Memory Model Definitions. For each model, we list the memory accesses and a list of dynamic rewrite rules.

3.4 Memory Models

We now formally specify the effects of memory models on programs. Memory model effects are similar to program transformations insofar they locally transform the program. However, the transformations are of a *dynamic* nature. Transformations may depend on information not statically known such as what actual address is accessed, or how many iterations a loop performs.[5]

To understand memory model effects, we visualize an executing program as an *event stream*, and the memory model as a component that tampers with the event stream by swapping or eliminating events, in an attempt to improve performance under the covers (but sometimes causing observable anomalies). Fig. 10 defines the set of events. There are memory events (loads, stores, atomic load-stores, and fences) as well as externally visible events (get, print). Each event contains all the relevant info, such as which variable was accessed, and what value(s) were loaded, stored, received, or printed. Moreover, all memory events carry a qualifier $h \in \{R, L, W, S\}$ to distinguish between different access types. On the hardware (x86) level, we use R and L to represent regular and locked accesses; on the language level, we use W and S to distinguish between weak (normal) and strong (volatile) accesses.

Each concurrent component produces a separate event stream which then gets merged (interleaved) with the others. Streams may be modified by the memory model both before and after they are merged. Only the final event stream is required to be value-consistent (each load sees the latest value stored to the same address). See Fig. 12 for an example.

Exactly what modifications are possible depends on the memory model; we define several useful models using rewrite rules in Fig. 11, to be discussed in the following subsections.

3.4.1 The (SC) Model

Sequential consistency is the most conservative memory model. It does not alter the event streams in any way, thus does not contain any rewrite rules.

3.4.2 The $(X86)$ Model

This model is our formalization of the memory model released by Intel in 2007 [10]. It features snooping store buffers, reflected by the rules **srsrl**[6] and **ersrlf**.[7] It provides full sequential consistency for all locked operations,[8] guarantees release semantics for stores,[9] and does not reorder loads with other loads.[10] Fences do not participate in rewriting and thus enforce ordering between preceding and succeeding events.

To the best of our knowledge, our specification is equivalent to the (informally specified) official memory model. For further confirmation, we have ascertained that our model gives the correct results on all 10 litmus tests in the official whitepaper [10]. Moreover, we were pleased to discover that it can reproduce the mysterious IRIW example (Fig. 12) in an elegant way (that is, without having to split the store into components for each processor). Essentially, we can explain the IRIW effect by applying store-load forwarding (the **ersrlf** rule) on the level of intermediate shared caches. The AMD architecture specification [1] explicitly allows processors to exhibit IRIW.

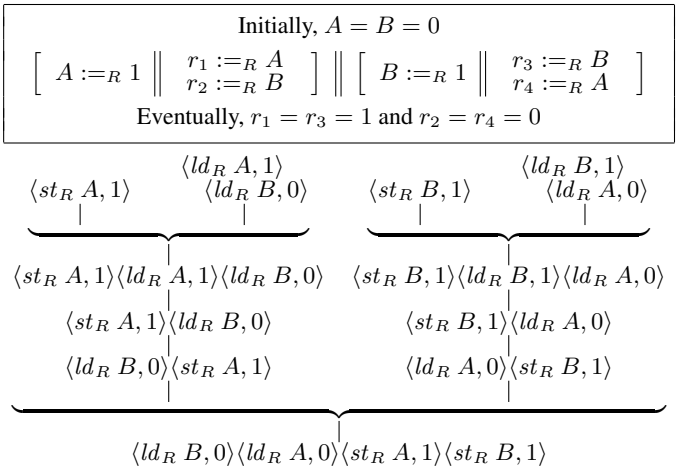

Figure 12. The IRIW example (in the box) is not prohibited on the X86 memory model (IRIW stands for independent reads of independent writes). Note that the parallel composition is not flat but hierarchical, reflecting a nested cache hierarchy where near processors can see updates before far processors. The diagram below the box shows how to derive this execution using the rewriting rules of the memory model *X86* (defined in Fig. 11). At the top, each processor emits an event sequence consistent with the program (the loaded values are justified later). Next, adjacent processors' instruction streams get interleaved. Then, we apply **ersrlf** on both sides to hide the snooped load. Next, we apply **srsrl** on both sides to delay the store past the load. Finally, we interleave the sequences and obtain a value-consistent sequence, which justifies the loaded values.

3.4.3 The (CLR_{blog}) Model

This model, to our knowledge, is the first formal specification of the working CLR memory model [3, 8, 16]. Events in this model consist of weak (regular) and strong (volatile) accesses. Both weak and strong stores have release semantics as indicated by the **ssl** rule that allows reordering of stores past loads. By the **swll** rule, loads can be reordered, except that strong loads have acquire semantics. Finally, **eswlf** allows weak loads to be eliminated with forwarded values from preceding stores.

3.4.4 The (CLR^{SC}_{blog}) Model

This model is a stronger variation of the (CLR_{blog}) model that ensures sequential consistency for strong accesses. This is reflected by all rules applying to weak accesses only (strong accesses do not participate in any rewriting).

4. Solution

In this section, we give a detailed account of our methodology to prove the soundness of a local transformation $(s \to s')$.

Our strategy is as follows. First, we characterize the set $\mathcal{B}$ of *behaviors* of a program snippet s (akin to *observations* of a program p) such that $\mathcal{B}$ captures effects of s that are observable by the rest of the program. We define a semantic bracket $[\![]\!]_M$ to recursively

[5] In fact, each individual loop iteration may be transformed differently, and dynamic transformations may easily cross loop and procedure boundaries.

[6] The rule **srsrl** expresses that the memory model may swap two consecutive events in the stream if the first one is a regular store, the second one is a regular load, and if they access different locations (expressed by the condition $L \neq L'$). This reflects the behavior of a store buffer: while a store is still in the buffer, the processor may execute a subsequent load to a different address.

[7] The rule **ersrlf** expresses that if a store is followed by a load from the same address that loads the same value, then the load may be hidden. This expresses snooping of the buffer: the load is satisfied directly from the buffer and not globally visible.

[8] None of the rewrite rules contain locked operations, which implies that they are interleaved in a sequentially consistent way.

[9] None of the rewrite rules move any events past a store.

[10] There is no rewrite rule for swapping loads.

compute the behaviors of s from its syntax for a given memory model M. Then we prove two key theorems.

The first theorem is based on the simple intuition that if s' does not introduce any new behaviors over s, then the effect of the transformation $(s \rightarrow s')$ is not observable. Specifically, it states that $[\![s']\!]_M \subseteq [\![s]\!]_M$ implies soundness. For example, suppose s' is obtained from s by reordering loads, and M reorders loads as well. Then $[\![s]\!]_M$ contains the reordered trace and thus $[\![s']\!]_M \subseteq [\![s]\!]_M$ which implies soundness.

Sometimes, transformations do introduce new behaviors but are still sound because the difference is not observable; for example, they may introduce irrelevant reads (see Fig. 16 at the end for some examples). We extend our methodology to handle this case by formalizing the notion of *procrastinable* transformations. Our generalized soundness theorem shows that if D is a set of procrastinable transformations for M, then $[\![s']\!]_{M \cup D} \subseteq [\![s]\!]_{M \cup D}$ implies soundness. This allows us to prove the soundness of more transformations.

4.1 Behaviors

To capture the behaviors of snippets, we need to capture (1) the effect on the local state, and (2) the sequence of events. Specifically, we let Q be the set of local states, defined as functions $\mathcal{R} \rightarrow \mathcal{X}$, and we let Evt be the set of events as defined in Fig. 10. Then we define the set of *behaviors*

$$\mathcal{B} = (Q \times Q \times Evt^*) \cup (Q \times Evt^\infty)$$

A triple (q, q', w) represents a terminating behavior that starts in local state q, ends in local state q', and emits the event sequence w. A pair (q, w) represents a nonterminating behavior that starts in local state q and emits the (finite or infinite) event sequence w.

For a set $B \subseteq \mathcal{B}$ and states $q, q' \subseteq Q$ we define the projections $[B]_{qq'} = \{w \mid (q, q', w) \in B\}$ and $[B]_q = \{w \mid (q, w) \in B\}$.

4.2 Dynamic Rewrite Rules

To define dynamic transformations of event sequences (as performed by the hardware), we use *rewrite rules* of the form $p \xrightarrow{\varphi} q$ where p and q are patterns and φ is an (optional) formula describing conditions under which the rule applies. We let $\mathcal{T}$ be the set of all such rewrite rules.

For example, consider the rewrite rules in Fig. 11 and Fig. 14. The rule **ssl** expresses that a store can be moved past a load that immediately follows it and accesses a different variable. The rule $\mathbf{edl}_h$ means that if a sequence contains two identical loads (targeting the same variable and loading the same value), one of them can be eliminated.

To use our generalized soundness theorem we need rules that involve multiple behaviors at a time. For instance, the rules **eil** and **iil** (Fig. 14) feature the wildcard character $*$. They do not rewrite individual sequences, but *sets* of sequences. The rule **eil** allows us to eliminate a load from a sequence if it appears for all possible values. The rule **iil** introduces a load into a sequence, once for each value. We call rules that feature wildcards *aggregate rewrite rules*, and all others *simple rewrite rules*. We now define the effects of a general rewrite rule more formally.

DEFINITION 3. *For a rewrite rule $t = p \xrightarrow{\varphi} q$, let $G_t \subset \mathcal{P}(Evt^*) \times \mathcal{P}(Evt^*)$ be the set of set pairs (S_1, S_2) such that there exists a valuation of the variables in p, q that satisfies φ and such that S_1 and S_2 correspond to all values (under varying assignments to the wildcards) of p and q, respectively. Then, define the operator $t : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$ by*

$$t(A) = \bigcup \{ w S_2 w' \mid w, w' \in Evt^* \\ (S_1, S_2) \in G_t \text{ and } w S_1 w' \subseteq A \}$$

For example, $G_{\mathbf{eil}}$ contains all set pairs (S_1, S_2) such that there exists a $l \in \mathcal{L}$ such that $S_1 = \{\langle ld\ l, x\rangle \mid x \in \mathcal{X}\}$ and $S_2 = \{\epsilon\}$. For simple rewrite rules t, all set pairs in G_t are of the form $(\{w_1\}, \{w_2\})$ for some $w_1, w_2 \in Evt^*$, and the definition above reduces to "normal" rewriting.

For a set $T \subset \mathcal{T}$, we define $T(A) = A \cup \bigcup_{t \in T} t(A)$. The following definition is useful for parallel rewriting.

DEFINITION 4. *Let $f : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$. Then we define the operators $P_f : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$ and $\widehat{P_f} : \mathcal{P}(Evt^\infty) \rightarrow \mathcal{P}(Evt^\infty)$ by*

$$P_f(A) = \bigcup \{ f(A_1) \cdots f(A_n) \mid \\ A_i \subset Evt^* \text{ such that } A_1 \cdots A_n \subseteq A \}$$

$$\widehat{P_f}(\hat{A}) = \bigcup \{ f(A_1) f(A_2) f(A_3) \cdots \mid \\ A_i \subset Evt^* \text{ such that } A_1 A_2 A_3 \cdots \subseteq \hat{A} \}.$$

Note that $\widehat{P_f}(\hat{A})$ may contain infinite sequences even if $\hat{A}$ does not.[11] We now show how to construct fixpoints for the effect of memory models $M \subseteq \mathcal{T}$ on behaviors.

DEFINITION 5. *A* memory model *is a finite set $M \subset \mathcal{T}$ of simple rewrite rules.*

DEFINITION 6. *Let M be a memory model. We define $M^* : \mathcal{P}(Evt^*) \rightarrow \mathcal{P}(Evt^*)$ and $M^\infty : \mathcal{P}(Evt^\infty) \rightarrow \mathcal{P}(Evt^\infty)$ by*

1. $M^*(A) = \bigcup_{k \geq 0} M^k(A)$
2. $M^\infty(\hat{A}) = \bigcup_{k \geq 0} (\widehat{P_{M^*}})^k(\hat{A})$

Moreover, for a set $B \subseteq \mathcal{B}$ of behaviors, define the closure B^M *as*

$$B^M = \{(q, q, w) \mid q, q' \in Q \text{ and } w \in M^*([B]_{qq'})\} \\ \cup \{(q, w) \mid q \in Q \text{ and } w \in M^\infty([B]_q)\}.$$

This definition reflects the effects of the memory model on behaviors. [12]. We can show that $(B^M)^M = B^M$ (see the appendix for a proof; the proof depends on M containing only simple rules).

4.3 Denotational Semantics

Fig. 13 defines the semantic bracket $[\![.]\!]_M : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{B})$ which assigns to each snippet s the set of behaviors $[\![s]\!]_M$ that s may exhibit on memory model M. The bracket $[\![.]\!]_M$ is defined recursively; it computes behaviors of snippets from the inside out, applying the memory model rewrite rules at each step. Sequential composition appends the behaviors of its constituents, while parallel composition interleaves them. The behaviors of a load include all possible values it could load (because the actual value depends on the context which is not known at this point). Later, at the level of the shared variable declaration, we filter out all behaviors that are not value-consistent.

[11] for example, consider $\hat{A} = \{\epsilon\}$ and $M = \{\epsilon \rightarrow 0\}$. Then $\widehat{P_M}(\hat{A})$ contains the infinite sequence $000\cdots$.

[12] Care is required to avoid undesired behaviors. For example, consider the rule $\mathbf{ssl}_h$ in Fig. 14 which represents the effect of stores being delayed in a buffer; while there is no bound on how long stores can be delayed, they must be eventually performed. Our formalism reflects this properly, as follows (using digits 0,1 instead of load and store events for illustration purposes). Let $A = \{1010\ldots\}$ and $T = \{10 \rightarrow 01\}$. Then $0^k1010\ldots$ is in $T^\infty(A)$, but $000\cdots$ is not.

Notations used. For $q \in Q$, $r \in \mathcal{R}$ and $x \in \mathcal{X}$ we let $q[r \mapsto x]$ denote the function that maps r to x, but is otherwise the same as the function q. For a shared variable $L \in \mathcal{L}$, let $Evt(L) \subseteq Evt$ be the set of memory accesses to L. For two sequences $w \in Evt^*$ and $w' \in Evt^\infty$, we let $ww' \in Evt^\infty$ be the concatenation as usual. For a sequence of finite sequences $w_1, w_2, \cdots \in Evt^*$, we let $w_1 w_2 \cdots \in Evt^\infty$ be the concatenation (which may be finite or infinite). We lift concatenation to sets of sequences as usual (elementwise): for example, for $S \subseteq Evt^*$ and $S' \subseteq Evt^\infty$ we let $SS' = \{ss' \mid s \in S, s \in S'\}$. For $w \in Evt^\infty$ and $i \in \mathbb{N}$, let $w[i] \in Evt$ be the event at position i (starting with 1). Let $dom\ w \subseteq \mathbb{N}$ be the set of positions of w. For two sequences $w, w' \in Evt^\infty$ we define the set of fair interleavings $(w \# w') \subseteq Evt^\infty$ to consist of all sequences $u \in Evt^\infty$ such that there exist strictly monotonic functions $f : dom\ w \to dom\ u$ and $g : dom\ w' \to dom\ u$ satisfying $rg\ f \cap rg\ g = \emptyset$ and $rg\ f \cup rg\ g = dom\ w$, and such that $w[i] = u[f(i)]$ and $w'[i] = u[g(i)]$ for all valid positions i. Note that the interleaving operator # is commutative and associative. For a subset of events $C \subseteq Evt$, we define the projection function $proj_C : Evt^\infty \to Evt^\infty$ to map a sequence to the largest subsequence containing only events in C. We write $proj_{-L}$ short for the function $proj_{Evt \setminus Evt(L)}$ (which removes all accesses to L). We call a sequence $w \in Evt^\infty$ *value-consistent* with respect to a shared variable $L \in \mathcal{L}$ and an initial value $x \in \mathcal{X}$ if for each load of L appearing in w, the value loaded matches the value of the rightmost store to L that precedes the load in w, or the initial value x if there is no such store. We let $Cons(L, x) \subseteq Evt^\infty$ be the set of all sequences that are value-consistent with respect to L and x. Similarly, we let $Cons(L, x, x') \subseteq Evt^*$ be the set of finite sequences that are value-consistent with respect to initial and final values x and x' of L, respectively.

4.4 Monotonicity of Observations

The following definition and theorem lay the foundation for our soundness proofs. For the proof, see the appendix.

DEFINITION 7. *Given a program p and a memory model M, define the set of observations as follows:*

$$\begin{aligned} obs_M(p) \quad = \quad & \{u \mid \exists (q, q', w) \in [\![p]\!]_M : u = proj_{Ext}(w)\} \\ & \cup\ \{\nabla u \mid \exists (q, w) \in [\![p]\!]_M : u = proj_{Ext}(w)\} \end{aligned}$$

THEOREM 8 (Monotonicity). *Let $M \subseteq \mathcal{T}$ be a memory model, and let $s, s' \in \mathcal{S}$ be two snippets such that $[\![s']\!]_M \subseteq [\![s]\!]_M$. Then, for any context c, we have $obs_M(c[s']) \subseteq obs_M(c[s])$.*

4.5 Main Soundness Theorems

THEOREM 9 (Simple Soundness). *Let $(s \to s')$ be a local transformation, and let $M \subseteq \mathcal{T}$ be a memory model. Then the following condition is sufficient to guarantee soundness:*

$$[\![s']\!]_M \subseteq [\![s]\!]_M .$$

PROOF. Let c be a context such that $p = c[s]$ and $p' = c[s']$ have no free variables. By Thm. 8, $[\![s']\!]_M \subseteq [\![s]\!]_M$ implies $obs_M(p') \subseteq obs_M(p)$. Thus, the transformation $p \Rightarrow p'$ is sound. □

We now proceed to the more powerful generalized soundness theorem which allows s' to introduce new behaviors as long as we can prove that they do not introduce additional observable behaviors.

DEFINITION 10. *Let $M \subseteq \mathcal{T}$ be a memory model, and let $D \in \mathcal{T}$ be a set of rewrite rules. We call D* procrastinable *on M if it satisfies the following conditions:*

1. *$m(P_D(A)) \subseteq P_D(M^*(A))$ for all $m \in M$ and $A \subseteq Evt^*$.*
2. *if $(S_1, S_2) \in G_d$ for some $d \in D$, then all sequences in S_2 are of length 0 or 1.*
3. *if $(S_1, S_2) \in G_d$ for some $d \in D$, and $w_2 \in S_2 \cap Cons(L, x, x')$, then there exists a $w_1 \in S_1 \cap Cons(L, x, x')$ such that $proj_{-L}(\{w_2\}) \in D(proj_{-L}(\{w_1\}))$.*
4. *if $(S_1, S_2) \in G_d$ for some $d \in D$, then $proj_{Ext}(S_2) \subseteq proj_{Ext}(S_1)$.*

We call a single transformation $d \in \mathcal{T}$ procrastinable if $\{d\}$ is procrastinable.

$\mathbf{edl}_h$ (eliminate double load)	=	$\langle ld_h\ L, x\rangle\langle ld_h\ L, x\rangle \to \langle ld_h\ L, x\rangle$
$\mathbf{eds}_h$ (eliminate double store)	=	$\langle st_h\ L, x\rangle\langle st_h\ L, x'\rangle \to \langle st_h\ L, x'\rangle$
$\mathbf{ecs}_h$ (eliminate confirmed store)	=	$\langle ld_h\ L, x\rangle\langle st_h\ L, x\rangle \to \langle ld_h\ L, x\rangle$
$\mathbf{eslf}_h$ (eliminate store-load forward)	=	$\langle st_h\ L, x\rangle\langle ld_h\ L, x\rangle \to \langle st_h\ L, x\rangle$
$\mathbf{eil}_h$ (eliminate irrelevant load)	=	$\langle ld_h\ L, *\rangle \to \epsilon$
$\mathbf{iil}_h$ (invent irrelevant load)	=	$\epsilon \to \langle ld_h\ L, *\rangle$

Figure 14. A list of procrastinable rewrite rules (Thm. 12).

THEOREM 11 (Generalized Soundness). *Let $(s \to s')$ be a local transformation, and let $M \subseteq \mathcal{T}$ be a memory model. Then the following condition is sufficient to guarantee soundness: there exists a procrastinable set $D \subseteq \mathcal{T}$ and a $k \geq 0$ such that for all $q, q' \in Q$, we have*

$$\left[[\![s']\!]_M\right]_{qq'} \subseteq (P_D)^k \left[([\![s]\!]_M)\right]_{qq'} \text{ and } \left[[\![s']\!]_M\right]_q \subseteq (\widehat{P_D})^k \left[[\![s]\!]_M\right]_q .$$

To use the generalized soundness theorem, we need to find suitable procrastinable sets D. The following theorem does just that.

THEOREM 12. *Of the dynamic rewrite rules shown in Fig. 14, the rules $\mathbf{edl}_S$, $\mathbf{eds}_S$, $\mathbf{ecs}_S$, $\mathbf{eslf}_S$, $\mathbf{eil}_S$, and $\mathbf{iil}_S$ are procrastinable on SC, the rules $\mathbf{edl}_R$, $\mathbf{eds}_R$, $\mathbf{eil}_R$, and $\mathbf{iil}_R$ are procrastinable on X86, and the rules $\mathbf{edl}_W$, $\mathbf{eds}_W$, $\mathbf{eil}_W$, and $\mathbf{iil}_W$ are procrastinable on CLR_{blog} and CLR^{SC}_{blog}.*

We give a rough description of the proof procedure here (we include a detailed proof in the appendix). Checking the properties 1 through 4 is largely mechanical. To check property 1, we need to examine the right-hand sides of parallel applications of elements in D and see in what ways they may overlap with the left-hand side of an individual rule in M. For each such overlap, we need to show that we can delay the parallel application of D past the application of M^*. Property 2 is immediate. Property 3 is straightforward. Property 4 is trivial as none of the rewrite rules involves any external events.

5. Tool

Our tool is written in F# and uses the automated theorem prover Z3 [7]. It takes as input a local transformation $(s \to s')$ and a memory model M. The snippets s, s' are specified using a sugared syntax; we currently support loop-free snippets without parallel composition only. The model M is specified by selecting a subset of the rewrite rules in Fig. 11 and Fig. 14.

The user then chooses one of two modes (verification or falsification) which take some optional additional input and operate as follows:

- In verification mode, the tool executes s and s' symbolically and attempts to prove $\forall q : \forall q' : [[\![s']\!]_M]_{qq'} \subseteq D^* [[\![s]\!]_M]_{qq'}$,

$$
\begin{array}{rcl}
[\![\mathbf{skip}]\!]_M & = & \{(q,q,\epsilon) \mid q \in Q\}^M \\
[\![r :=_h L]\!]_M & = & \{(q,q[r \mapsto x],\langle ld\ L, x\rangle) \mid q \in Q, x \in \mathcal{X}\}^M \\
[\![L :=_h r]\!]_M & = & \{(q,q,\langle st\ L, q(r)\rangle \mid q \in Q\}^M \\
[\![r_0 := f(r_1 \ldots r_n)]\!]_M & = & \{(q,q[r_0 \mapsto f(q(r_1)\ldots q(r_n)],\epsilon) \mid q \in Q\}^M \\
[\![r_r := cas_h(L, r_c, r_n)]\!]_M & = & \left(\begin{array}{l} \{(q,q[r_r \mapsto q(r_c)],\langle ldst\ L, q(r_c), q(r_n)\rangle) \mid q \in Q\} \\ \cup\, \{(q,q[r_r \mapsto x],\langle ldst\ L, x, x\rangle) \mid q \in Q, x \in \mathcal{X}, x \neq q(r_c)\} \end{array}\right)^M \\
[\![get\ r]\!]_M & = & \{(q,q[r \mapsto x],\langle get\ x\rangle) \mid q \in Q, x \in \mathcal{X}\}^M \\
[\![print\ r]\!]_M & = & \{(q,q,\langle print\ q(r)\rangle \mid q \in Q\}^M \\
[\![s_1; s_2]\!]_M & = & \left(\begin{array}{l} \{(q,q',w) \mid \text{there exist } (q,q'',w_1) \in [\![s_1]\!]_M \text{ and } (q'',q',w_2) \in [\![s_2]\!]_M \text{ such that } w = w_1 w_2\} \\ \cup\, \{(q,w) \mid (q,w) \in [\![s_1]\!]_M\} \\ \cup\, \{(q,w) \mid \text{there exist } (q,q',w_1) \in [\![s_1]\!]_M \text{ and } (q',w_2) \in [\![s_2]\!]_M \text{ such that } w = w_1 w_2\} \end{array}\right)^M \\
[\![s_1 \parallel \cdots \parallel s_n]\!]_M & = & \left(\begin{array}{l} \{(q,q',w) \mid \text{there exist } (q,q_i,w_i) \in [\![s_i]\!]_M \text{ for all } 1 \leq i \leq n \text{ such that} \\ \qquad w \in w_1 \# \ldots \# w_n \text{ and such that } q'(r) = q_i(r) \text{ for all } r \in FV(s_i) \text{ and} \\ \qquad q'(r) = q(r) \text{ for all } r \notin FV(s_1) \cup \ldots FV(s_n)\} \\ \cup\, \{(q,w) \mid \text{ there exist } w_1,\ldots,w_n \in Evt^\infty \text{ and a nonempty subset } D \subseteq \{1,\ldots,n\} \\ \qquad \text{such that for all } j \in D, \text{ we have a behavior } (q,w_j) \in [\![s_j]\!]_M, \\ \qquad \text{and for all } j \notin D, \text{ we have a behavior } (q,q_j,w_j) \in [\![s_j]\!]_M \text{ for some } q_j, \\ \qquad \text{and } w \in w_1 \# \ldots \# w_n\} \end{array}\right)^M \\
[\![\mathbf{if}\ r\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]_M & = & \left(\begin{array}{l} \{(q,q',w) \mid (q(r) \neq 0 \wedge (q,q',w) \in [\![s_1]\!]_M) \vee (q(r) = 0 \wedge (q,q',w) \in [\![s_2]\!]_M)\} \\ \cup\, \{(q,w) \mid (q(r) \neq 0 \wedge (q,w) \in [\![s_1]\!]_M) \vee (q(r) = 0 \wedge (q,w) \in [\![s_2]\!]_M)\} \end{array}\right)^M \\
[\![\mathbf{while}\ r\ \mathbf{do}\ s]\!]_M & = & \left(\begin{array}{l} \{(q_0,q_n,w_1\cdots w_n) \mid \text{there exist } n \geq 0 \text{ and } q_0,\ldots,q_n \text{ such that } (q_i,q_{i+1},w_{i+1}) \in [\![s]\!]_M \\ \qquad \text{for } 0 \leq i < n, \quad \text{and } q_0(r) \neq 0,\ldots,q_{n-1}(r) \neq 0, \text{ and } q_n(r) = 0\} \\ \cup\, \{(q_0,w_1w_2\cdots) \mid \exists q_1,q_2,\ldots : (q_i,q_{i+1},w_{i+1}) \in [\![s]\!]_M \text{ and } q_i(r) \neq 0\} \\ \cup\, \{(q_0,w_1\cdots w_n) \mid \text{there exist } n \geq 1 \text{ and } q_0,\ldots,q_{n-1} \text{ such that } q_i(r) \neq 0 \text{ for all } i \text{ and} \\ \qquad (q_i,q_{i+1},w_{i+1}) \in [\![s]\!]_M \text{ for } 0 \leq i < n-1 \text{ and } (q_{n-1},w_n) \in [\![s]\!]_M\} \end{array}\right)^M \\
[\![\mathbf{local}\ L = x\ \mathbf{in}\ s]\!]_M & = & \left(\begin{array}{l} \{(q,q',w) \mid \text{ there exists a behavior } (q[r \mapsto x],q'',w) \in [\![s]\!]_M \\ \qquad \text{such that } q' = q''[r \mapsto q(r)]\} \\ \cup\, \{(q,w) \mid \text{ there exists a behavior } (q[r \mapsto x],w) \in [\![s]\!]_M\} \end{array}\right)^M \\
[\![\mathbf{share}\ L = x\ \mathbf{in}\ s]\!]_M & = & \left(\begin{array}{l} \{(q,q',w) \mid \text{ there exists a behavior } (q,q',w') \in [\![s]\!]_M \\ \qquad \text{such that } w' \in Cons(L,x) \text{ and } w = proj_{\_L}(w')\} \\ \cup\, \{(q,w) \mid \text{ there exists a behavior } (q,w') \in [\![s]\!]_M \\ \qquad \text{such that } w' \in Cons(L,x) \text{ and } w = proj_{\_L}(w')\} \end{array}\right)^M
\end{array}
$$

Figure 13. Denotational Semantics of our Calculus, parameterized by a set M of dynamic rewrite rules. An empty set M represents the standard semantics (sequential consistency).

where D is an optionally specified set of procrastinable rules, or empty by default[13]. If successful, soundness is established (by Thm. 11). Otherwise, the theorem prover will attempt to find a behavior in the set difference and report it to the user.

- In falsification mode, the tool takes a context c (which may contain several threads) as an additional input. It then computes the closure of $c[s']$ and $c[s]$ under interleavings and under M, and solves for a behavior of $c[s']$ that is not observationally equivalent (assuming that *all* initial and final values of all variables are being observed) to any behavior in $c[s]$. If such a behavior is found, soundness has been successfully refuted. Otherwise, the result is inconclusive.

Both modes are semi-automatic in the following sense. If the verification fails, it is up to the user to add more procrastinable rules or to try falsification instead. If falsification fails, it is up to the user to change the context or to try verification instead. Moreover, the tool does not yet verify whether the procrastinable rules D specified by the user are truly procrastinable; we still use manual proofs for this (see Thm. 12). Future work may further automate this process.

[13] If irrelevant-load-introduction is one of the rules, we can not compute D^* and will use D^k instead, for a manually specified k. This does not compromise the validity of a successful soundness proof, but it may mean that we can not prove some rules sound if the bound is too low.

5.1 Symbolic computation

Internally, the tool represents sets of behaviors by functions $ESeq \to \mathcal{F}$, where $ESeq$ is the set of event sequences (event names only, not containing values) and $\mathcal{F}$ is a set of formulas. The meaning is that for each sequence of events the corresponding formula expresses the conditions under which that sequence is in the set, by constraining the variables L_i and x_i (representing the target variable and the loaded/stored value of the i-th event in the sequence) as well as $\mathcal{R}$ and $\mathcal{R}'$ (representing initial and final values of local variables, respectively). For example, for shared variables $\mathcal{L} = \{A, B\}$ and local variables $\mathcal{R} = \{r, s\}$ we represent the set $[\![r :=_h A; \mathbf{if}\ r\ \mathbf{then}\ r :=_h B]\!]$ by the function which maps all sequences to false except for

$$
\begin{array}{rcl}
\langle ld_h\rangle & \mapsto & (L_1 = A) \wedge (x_1 = 0) \wedge (r' = x_1) \wedge (s' = s) \\
\langle ld_h\rangle\langle ld_h\rangle & \mapsto & (L_1 = A) \wedge (x_1 \neq 0) \wedge (L_2 = B) \\
 & & \wedge (r' = x_2) \wedge (s' = s).
\end{array}
$$

transformation name (see Fig. 16)	SC	$X86$	CLR_{blog}	CLR^{SC}_{blog}	rules
(load reordering)	×	×	√	√	
(store reordering)	×	×	×	×	
(irrelevant read elim.)	√	√	√	√	**eil**
(irrelevant read intr.)	√	√	√	√	**iil**
(red. read-after-read elim.)	√	√	√	√	**edl**
(red. read-after-wr. elim.)	√	√	√	√	**eslf** on SC
(red. wr.-bef.-wr. elim.)	√	√	√	√	**eds**
(red. wr.-after-read elim.)	√	×	×	×	**ecs**
(JIT example 1)	n/a	n/a	×	×	
(JIT example 2)	n/a	n/a	×	×	
(JIT example 3)	n/a	n/a	×	×	

Figure 17. Soundness results for the examples from Fig. 16. For sound transformations (marked by √), we list any additional invisible rules required for the proof on the right. For unsound transformations (marked by ×), we show example executions in Fig. 18. All results were validated by our tool.

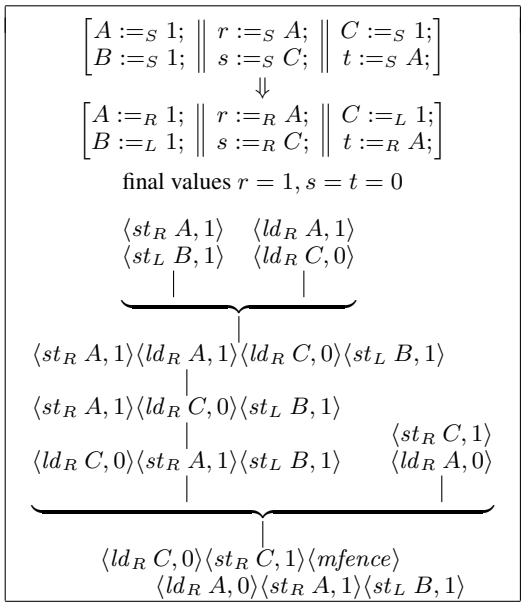

Figure 19. Derivation for the JIT example 3. We show the original program, the transformed program, and an execution of the transformed program that is not possible on the original program. All shared variables and registers are initially zero.

For finite sets, all but finitely many sequences are mapped to false. We can thus easily represent such sets in our tool implementation. Moreover, we can perform set union as elementwise disjunction, and we can apply rewrite rules symbolically, as shown in Fig. 15. Using the automated theorem prover, we can prove/refute inclusion of two symbolic sets $\Phi \subseteq \Phi'$ by proving/refuting $\bigwedge_s(\Phi(s) \Rightarrow \Phi'(s))$. This allows us to compute $M^*(\Phi)$ by repeatedly applying the rewrite rules until a fixpoint is reached.

6. Results

Using our tool, we successfully proved or refuted soundness of the 11 transformations in Fig. 16 for the four memory models defined in §3.4, with results indicated in Fig. 17. The tool performed acceptably, providing a good interactive experience. The total time needed by the tool to prove/refute all examples is about 2 minutes, with about 90% of the time spent on the last 4 examples.

The first eight are transformations that are commonly used by optimizing compilers; it was recently found that some of them are not sound for the Java memory model [18].

The first seven examples showed little surprise. The eighth one (redundant-write-after-read elimination) exhibited a slightly surprising behavior: while sound on SC, it is unsound on $X86$, CLR_{blog} and CLR^{SC}_{blog} (Fig. 18 shows a falsification for CLR_{blog} and CLR^{SC}_{blog}; Fig. 20 shows a falsification for $X86$). This example contradicts the (false) intuition that weaker memory models always permit more transformations. In this case, the rule **ecs** (which we use for the soundness proof) is procrastinable on SC, but not on $X86$, CLR_{blog} and CLR^{SC}_{blog}, where it may interact with store-load and load-load reordering.

The JIT transformation examples proved particularly interesting. Example 1 shows that regular X86 loads do *not* guarantee acquire semantics even though the processors prohibit reordering of loads with each other. The reason is store-load-forwarding: once forwarded from a store, a load may effectively move past a following load (Fig. 18). This discovery means that the current CLR JIT compiler for the x86 platform is not correct; it will be fixed in the future by strictly emitting locked instructions for all volatile stores, thereby also satisfying the stronger CLR^{SC}_{blog} model.

Example 2 shows that adding fences between volatiles is not always sufficient, because of store-load forwarding across processors (Fig. 18). This example points out a bug in the JSR-133 recommendations [12].

Example 3 shows that, when converting volatile loads to regular loads, all volatile stores need to be converted to locked stores, even if several volatile stores follow each other (compiler writers may be tempted to emit only the last volatile write as a locked instruction). The trace is shown in Fig. 19.

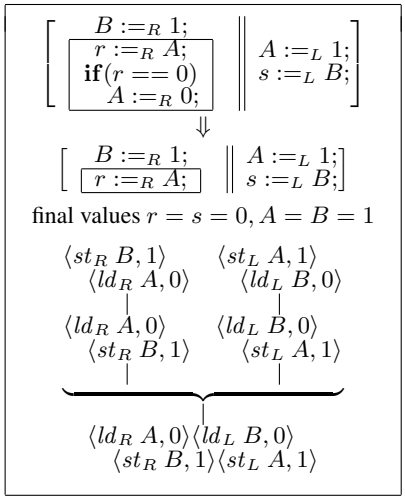

Figure 20. Refutation for the write-after-read elimination example on $X86$. We show the original program, the transformed program, and an execution of the transformed program that is not possible on the original program. All shared variables and registers are initially zero.

$$
\begin{array}{rcl}
(\mathbf{ssl}(\Phi))(s) & = & \bigvee \{\Phi(s') \wedge (L_{k+1} \neq L_{k+2}) \mid \exists w, w' : (s' = w\langle st_h\rangle\langle ld_{h'}\rangle w') \text{ and } (s = w\langle ld_{h'}\rangle\langle st_h\rangle w') \text{ and } k = |w|\} \\
(\mathbf{edl}_h(\Phi))(s) & = & \bigvee \left\{\Phi(s') \uparrow_{k+2}^{-1} \mid \exists w, w' : (s = w\langle ld_h\rangle w') \text{ and } (s' = w\langle ld_h\rangle\langle ld_h\rangle w') \text{ and } k = |w|\right\} \\
(\mathbf{eds}_h(\Phi))(s) & = & \bigvee \left\{\exists x.\, ((\Phi(s')[x/x_{k+1}]) \uparrow_{k+2}^{-1}) \mid \exists w, w' : (s = w\langle st_h\rangle w') \text{ and } (s' = w\langle st_h\rangle\langle st_h\rangle w') \text{ and } k = |w|\right\} \\
(\mathbf{eil}_h(\Phi))(s) & = & \bigvee \left\{\exists L.\forall x.\, ((\Phi(s')[x/x_{k+1}][L/L_{k+1}]) \uparrow_{k+2}^{-1}) \mid \exists w, w' : (s = ww') \text{ and } (s' = w\langle ld_h\rangle w') \text{ and } k = |w|\right\} \\
(\mathbf{iil}_h(\Phi))(s) & = & \bigvee \left\{\Phi(s') \uparrow_{k+1}^{1} \mid \exists w, w' : (s = w\langle ld_h\rangle w') \text{ and } (s' = ww') \text{ and } k = |w|\right\}
\end{array}
$$

Figure 15. Symbolic application of some or our rewrite rules from Fig. 11 and Fig. 14 (the remaining ones are analogous). In each case, Φ is a symbolic set, and s, s', w, w' are finite sequences. The notation $\varphi[x/y]$ represents the formula φ with occurrences of y substituted by x. The notation $\varphi \uparrow_c^d$ represents the formula φ with all occurrences of x_i, L_i where $i \geq c$ substituted by x_{i+d}, L_{i+d}, respectively.

(load reordering)	$\{\mathbf{if}\ r\ \mathbf{then}\ \{s := A;\ t := B\}\ \mathbf{else}\ \{t := B;\ s := A\}\} \rightarrow \{s := A;\ t := B\}$
(store reordering)	$\{\mathbf{if}\ r\ \mathbf{then}\ \{A := s;\ B := t\}\ \mathbf{else}\ \{B := t;\ A := s\}\} \rightarrow \{A := s;\ B := t\}$
(irrelevant read elimination)	$\{\mathbf{local}\ r = 0\ \mathbf{in}\ \{r := A;\ \mathbf{if}\ r\ \mathbf{then}\ \{B := s\}\ \mathbf{else}\ \{B := s\}\}\} \rightarrow \{B := s\}$
(irrelevant read introduction)	$\{\mathbf{if}\ r\ \mathbf{then}\ \mathbf{local}\ s = 0\ \mathbf{in}\ \{s := A;\ B := s\}\} \rightarrow \{\mathbf{local}\ s = 0\ \mathbf{in}\ \{s := A;\ \mathbf{if}\ r\ \mathbf{then}\ B := s\}\}$
(redundant read-after-read elim.)	$\{r := A;\ b := A\} \rightarrow \{r := A;\ b := r\}$
(redundant read-after-write elim.)	$\{A := r;\ s := A\} \rightarrow \{A := r;\ s := r\}$
(redundant write-before-write elim.)	$\{A := r;\ A := s\} \rightarrow \{A := s\}$
(redundant write-after-read elim.)	$\{r := A; \mathbf{if}\ r == 0\ \mathbf{then}\ A := 0\} \rightarrow \{r := A\}$
(JIT example 1)	replace all strong loads with X86 regular loads and all strong stores with X86 regular stores
(JIT example 2)	like JIT example 1, but insert a fence between any two strong accesses except if they are both loads
(JIT example 3)	$\{r :=_S A;\ s :=_S B\} \rightarrow \{r :=_R A;\ s :=_L B\}$

Figure 16. Example transformations. The snippets follow the syntax defined in §3.1, with $\mathcal{L} = \{A, B, \dots\}$ and $\mathcal{R} = \{r, s, t, \dots\}$, and assuming default access types of S for SC, R for $X86$, and W for CLR_{blog} and CLR_{blog}^{SC}.

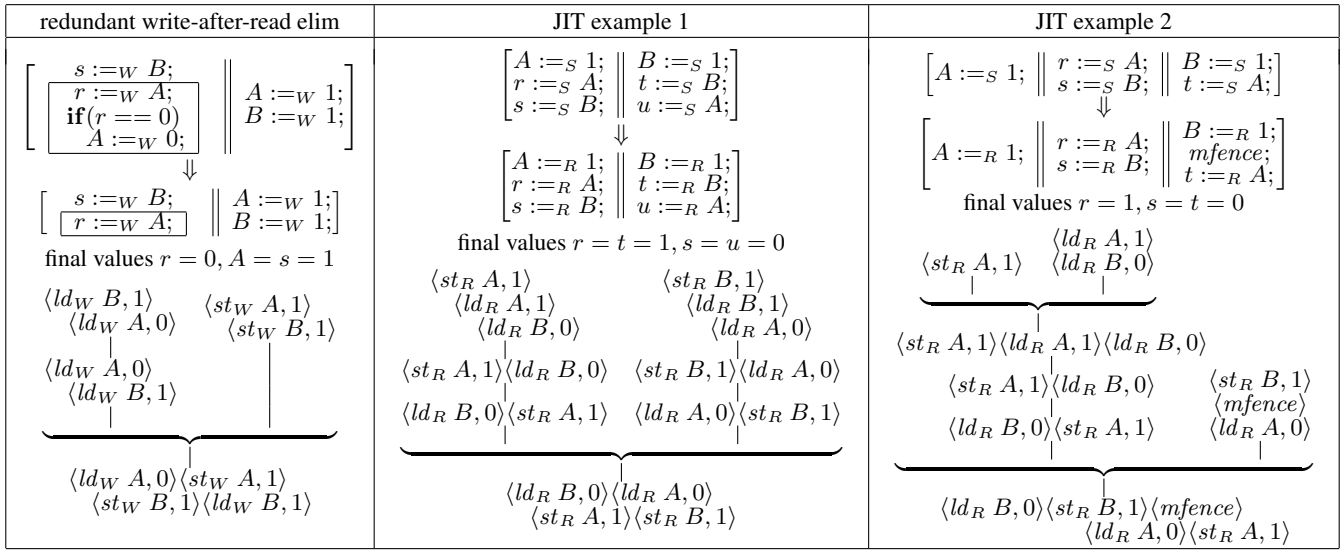

Figure 18. Derivations for the refuted examples. For each example, we show the original program, the transformed program, and an execution of the transformed program that is not possible on the original program. All shared variables and registers are initially zero.

7. Related Work

Our work is closely related to previous efforts on mechanical verification compiler transformations [19, 13, 11, 14] for sequential programs. Our work extends these efforts for concurrent programs.

Recent work has proposed language memory models [15, 2, 17] that enable a class of compiler optimizations, while providing an intuitive contract to the programmer. Our work is complementary and automates the soundness proofs of these optimizations for proposed memory models. In addition, our work also addresses the problem of translating language constructs into hardware primitives so as to retain the language memory model guarantees in the presence of hardware relaxations. Finally, our work was partly motivated by recent work [6, 18] that demonstrated the difficulty of manually verifying compiler optimizations against memory models.

Our work is also related to the work in verifying program behaviors against memory models [4, 5]. However, our focus on compiler transformations requires methodologies that allow local reasoning of program snippets.

Acknowledgments

We thank Jaroslav Sevcik, Peter Sewell, Doug Lea, Sarita Adve, Vijay Saraswat and Herb Sutter for enlightening discussions and comments on relaxed memory models.

References

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, September 2007.

[2] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Programming Language Design and Implementation (PLDI)*, pages 68–78, 2008.

[3] C. Brumme. cbrumme's weblog. `http://blogs.gotdotnet.com/cbrumme/archive/2003/05/17/51445.aspx`.

[4] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[5] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer-Aided Verification (CAV)*, pages 107–120, 2008.

[6] P. Cenciarelli and E. Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *In 16th European Symposium on Programming (ESOP)*, 2007.

[7] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[8] J. Duffy. Joe Duffy's Weblog. `http://www.bluebytesoftware.com/blog/2007/11/10/CLR20MemoryModel.aspx`.

[9] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[10] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*, August 2007.

[11] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[12] D. Lea. The jsr-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[13] S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *Programming Language Design and Implementation (PLDI)*, pages 220–231, 2003.

[14] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of programming languages (POPL)*, pages 42–54, 2006.

[15] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, 2005.

[16] V. Morrison. Understand the impact of low-lock techniques in multithreaded apps. *MSDN Magazine*, 20(10), October 2005.

[17] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP '07: Principles and practice of parallel programming*, pages 161–172, 2007.

[18] J. Sevcik and D. Aspinall. On validity of program transformations in the Java memory model. In *European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[19] W. D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, 1989.

A. Proofs

A.1 Observations

DEFINITION 13. *Define the function obs (similar to the previously defined function obs_M on programs, see Def. 7) as an overload of the following mappings:*

$$\begin{array}{ll} obs: & \mathcal{P}(Evt^*) \to \mathcal{P}(\mathcal{O}) \\ & obs(A) = \{w \mid w \in A\} \\ obs: & \mathcal{P}(Evt^\infty) \to \mathcal{P}(\mathcal{O}) \\ & obs(\hat{A}) = \{\nabla w \mid w \in \hat{A}\} \\ obs: & \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{O}) \\ & obs(B) = obs(\bigcup_{qq'} [B]_{qq'}) \cup obs(\bigcup_q [B]_q) \end{array}$$

LEMMA 14. *Let M be a memory model, and s a snippet. Then $obs([\![s]\!]_M) = obs_M(s)$.*

PROOF. directly from Def. 7 and Def. 13. □

A.2 Behavior Transformation Operators

To make the proofs more straightforward, we define behavior transformation operators that capture how the semantic bracket computes behavior sets, shown in Fig. 21. Each of them is a function $\mathcal{P}(\mathcal{B})^n \to \mathcal{P}(\mathcal{B})$ for some $n \geq 1$.

LEMMA 15. *For all $n \geq 1$, for all $s, s_1, s_2, \ldots s_n \in \mathcal{S}$, for all $F_1, \ldots, F_n \subseteq \mathcal{R}$, for all $x \in \mathcal{X}$, for all $r \in \mathcal{R}$ and for all $L \in \mathcal{L}$, we have*

$$\begin{array}{c} [\![s_1;\ s_2]\!]_M = \lambda_{seq}([\![s_1]\!]_M, [\![s_2]\!]_M)^M \\ [\![s_1 \ldots s_n]\!]_M = \lambda_{par,FV(s_1),\ldots,FV(s_n)}([\![s_1]\!]_M, \ldots, [\![s_n]\!]_M)^M \\ [\![\textbf{if } r \textbf{ then } s_1 \textbf{ else } s_2]\!]_M = \lambda_{cond,r}([\![s_1]\!]_M, [\![s_2]\!]_M)^M \\ [\![\textbf{while } r \textbf{ do } s]\!]_M = \lambda_{while,r}([\![s]\!]_M)^M \\ [\![\textbf{local } r = x \textbf{ in } s]\!]_M = \lambda_{local,r,x}([\![s]\!]_M)^M \\ [\![\textbf{share } L = x \textbf{ in } s]\!]_M = \lambda_{share,L,x}([\![s]\!]_M)^M \end{array}$$

PROOF. By straightforward comparison of the individual cases in Fig. 13 and Fig. 21. Essentially, we are just rewriting the set comprehensions of our earlier definition, using set union. Note that our notation assumes that vacuous set union or set concatenation operations (e.g. in $\lambda_{cond,r}$) produce the neutral elements $\emptyset$ and $\{\epsilon\}$, respectively. □

A.3 Monotonicity

DEFINITION 16. *Let S be some set, and $n \geq 0$. A function $f : \mathcal{P}(S)^n \to \mathcal{P}(S)$ is called* monotonous *if $n = 0$ or if for all $k \in \{1, \ldots, n\}$ and for all sets $S_1, S_2, \ldots, S_n \subseteq S$ and $S' \subseteq S$ we have*

$$S_k \subseteq S' \quad \Rightarrow \quad f(S_1, \ldots, S_n) \subseteq f(S_1, \ldots, S_{k-1}, S', S_{k+1}, \ldots, S_n)$$

A.4 Monotonicity Lemmas

LEMMA 17 (Union of Monotonic Functions). *Let S be some set, let $f : \mathcal{P}(S)^n \to \mathcal{P}(S)$ be monotonous, and let $S_i \subset S$ be a collection of sets where $i \in I$ with I an arbitrary index set. Then*

$$\bigcup_{i \in I} f(S_i) \subseteq f\left(\bigcup_{i \in I} S_i\right)$$

PROOF. Because $S_i \subseteq \bigcup_{j \in I} S_j$ for all i, we know (by monotonicity of f) that $f(S_i) \subseteq f\left(\bigcup_{j \in I} A_j\right)$ for all i. This implies the claim. □

LEMMA 18 (Monotonicity of Rewriting). *For $t \in \mathcal{T}$ and $T \subset \mathcal{T}$, the corresponding operators $\mathcal{P}(Evt^*) \to \mathcal{P}(Evt^*)$ are monotonous.*

PROOF. (Monotonicity of t). Let $A \subseteq A' \subseteq Evt^*$, and let $w \in t(A)$. By Def. 3, there exist $(S_1, S_2) \in G_t$ and $w_1, w_2 \in Evt^*$ such that $w \in w_1 S_2 w_2$ and $w_1 S_1 w_2 \subseteq A$. But then $w_1 S_1 w_2 \subseteq A'$ as well, therefore $w_1 S_2 w_2 \subseteq t(A')$ and thus $w \in t(A')$.

For $n \geq 1$, for $F_1, \ldots, F_n \subseteq \mathcal{R}$, for $x \in \mathcal{X}$, for $r \in \mathcal{R}$ and for $L \in \mathcal{L}$, define $\left\{ \begin{array}{rcl} \lambda_{\mathit{seq}} & : & \mathcal{P}(\mathcal{B})^2 \to \mathcal{P}(\mathcal{B}) \\ \lambda_{\mathit{par},F_1,\ldots,F_n} & : & \mathcal{P}(\mathcal{B})^n \to \mathcal{P}(\mathcal{B}) \\ \lambda_{\mathit{cond},r} & : & \mathcal{P}(\mathcal{B})^2 \to \mathcal{P}(\mathcal{B}) \\ \lambda_{\mathit{while},r} & : & \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B}) \\ \lambda_{\mathit{local},r,x} & : & \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B}) \\ \lambda_{\mathit{share},L,x} & : & \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B}) \end{array} \right\}$ as follows:

$$
\begin{array}{rcl}
[\lambda_{\mathit{seq}}(B_1,B_2)]_{qq'} & = & \bigcup_{p\in Q} [B_1]_{qp}\,[B_2]_{pq'} \\
[\lambda_{\mathit{seq}}(B_1,B_2)]_{q} & = & [B_1]_{q} \cup \bigcup_{p\in Q} [B_1]_{qp}\,[B_2]_{p} \\
[\lambda_{\mathit{par},F_1,\ldots,F_n}(B_1,B_2,\ldots,B_n)]_{qq'} & = & \left\{ \begin{array}{ll} \emptyset & \text{if not } \big(\forall r \in \bigcup_k F_k : q'(r) = q(r)\big) \\ \#_{1\leq k\leq n} \bigcup_{p\in P_k} [B_k]_{qp} & \text{otherwise, where } P_k = \{p \in Q \mid \forall r \in F_k : p(r) = q'(r)\} \end{array} \right. \\
[\lambda_{\mathit{par},F_1,\ldots,F_n}(B_1,B_2,\ldots,B_n)]_{q} & = & \bigcup_{D\subseteq\{1,\ldots,n\}} \Big(\big(\#_{k\in\{1,\ldots,n\}\setminus D} \bigcup_{p\in Q} [B_k]_{qp} \big) \,\#\, \big(\#_{k\in D} [B_k]_{q} \big) \Big) \\
[\lambda_{\mathit{cond},r}(B_1,B_2)]_{qq'} & = & \left\{ \begin{array}{ll} [B_1]_{qq'} & \text{if } q(r) \neq 0 \\ [B_2]_{qq'} & \text{if } q(r) = 0 \end{array} \right. \\
[\lambda_{\mathit{cond},r}(B_1,B_2)]_{q} & = & \left\{ \begin{array}{ll} [B_1]_{q} & \text{if } q(r) \neq 0 \\ [B_2]_{q} & \text{if } q(r) = 0 \end{array} \right. \\
[\lambda_{\mathit{while},r}(B)]_{qq'} & = & \bigcup_{\substack{q_0,\ldots q_n\in Q \\ n\geq 0,\ q_0=q,\ q_n=q', \\ q_n(r)=0,\ \forall i<n:\ q_i(r)\neq 0}} [B]_{q_0 q_1} \cdots [B]_{q_{n-1} q_n} \\
[\lambda_{\mathit{while},r}(B)]_{q} & = & \left(\bigcup_{\substack{q_0,q_1,q_2\cdots\in Q \\ q_0=q \\ \forall i:\ q_i(r)\neq 0}} [B]_{q_0 q_1}\,[B]_{q_1 q_2} \cdots \right) \cup \left(\bigcup_{\substack{q_0,\ldots q_{n-1}\in Q \\ n\geq 1, q_0=q \\ \forall i:\ q_i(r)\neq 0}} [B]_{q_0 q_1} \cdots [B]_{q_{n-2} q_{n-1}}\,[B]_{q_{n-1}} \right) \\
[\lambda_{\mathit{local},r,x}(B)\ ]_{qq'} & = & \left\{ \begin{array}{ll} \emptyset & \text{if } q'(r) \neq q(r) \\ \bigcup_{x'\in\mathcal{X}} [B]_{q[r\mapsto x]\,q'[r\mapsto x']} & \text{otherwise} \end{array} \right. \\
[\lambda_{\mathit{local},r,x}(B)\ ]_{q} & = & [B]_{q[r\mapsto x]} \\
[\lambda_{\mathit{share},L,x}(B)\ ]_{qq'} & = & \mathit{proj}_{-L}([B]_{qq'} \cap \mathit{Cons}(L,x)) \\
[\lambda_{\mathit{share},L,x}(B)\ ]_{q} & = & \mathit{proj}_{-L}([B]_{q} \cap \mathit{Cons}(L,x))
\end{array}
$$

Figure 21. Definition of the behavior transformation operators. These functions capture how the semantic bracket $[\![.]\!]$ computes behavior sets "from the inside out".

(Monotonicity of T). By definition, $T(A) = A \cup \bigcup_{t\in T} t(A)$. Therefore, T is a union of monotonous functions and thus also monotonous. □

LEMMA 19 (Monotonicity of Parallel Rewriting). *The parallel operators P_f and $\widehat{P_f}$ are always monotonous, for arbitrary $f : \mathcal{P}(Evt^*) \to \mathcal{P}(Evt^*)$.*

PROOF. (Monotonicity of P_f). Let $A \subseteq A' \subseteq Evt^*$, and let $w \in P_f(A)$. By Def. 4, there exist $A_1, \ldots, A_n \subset Evt^*$ such that $A_1 \cdots A_n \subseteq A$ and $w \in f(A_1) \cdots f(A_n)$. But then also $A_1 \cdots A_n \subseteq A'$ and therefore $w \in P_f(A')$.

(Monotonicity of $\widehat{P_f}$). Let $A \subseteq A' \subseteq Evt^*$, and let $w \in \widehat{P_f}(A)$. By Def. 4, there exist $A_1, A_2, \cdots \subset Evt^*$ such that $A_1 A_2 \cdots \subseteq A$ and $w \in f(A_1)f(A_2)\cdots$. But then also $A_1 A_2 \cdots \subseteq A'$ and therefore $w \in \widehat{P_f}(A')$. □

DEFINITION 20. *Let S be some set. A function $f : \mathcal{P}(S) \to \mathcal{P}(S)$ is called* expanding *if for all sets $S' \subseteq S$ we have $S' \subseteq f(S')$.*

LEMMA 21 (Parallel Rewriting Expands). *For any $D \subseteq \mathcal{T}$ the parallel operators P_D and $\widehat{P_D}$ are expanding.*

PROOF. By definition, for all sets $A \subseteq Evt^*$, we have $A \subseteq D(A)$. This easily implies the claim for P_D (choose $n = 1$ and $A_1 = A$ in Def. 4). For $\widehat{P_D}$, suppose $w \in \hat{A}$ for $\hat{A} \subseteq Evt^\infty$. Then we can decompose w into a concatenation of infinitely many singleton sets $\{w\} = \{w_1\}\{w_2\}\ldots$ (we can choose the w_i to become ϵ at some point if w is not infinite). Then $\{w_i\} \subseteq D(\{w_i\})$ and thus $\{w\} \subseteq \widehat{P_D}(\hat{A})$. □

LEMMA 22 (Monotonicity of Memory Models). *For any memory models $M, M_1, M_2 \subseteq \mathcal{T}$ and $B, B_1, B_2 \subseteq \mathcal{B}$ we have*

1. $B_1 \subseteq B_2 \;\Rightarrow\; B_1^M \subseteq B_2^M$

2. $M_1 \subseteq M_2 \Rightarrow B^{M_1} \subseteq B^{M_2}$

PROOF.

(Claim 22.1). By Lemma 18, the operator M is monotonous. Because finite compositions and arbitrary unions of monotonous operators are monotonous, by 6.1we know M^* is monotonous also. Now, by Lemma 19, we know $\widehat{P_{M^*}}$ is monotonous. Thus, by 6.2, M^∞ is monotonous (unions of finite compositions of monotonous functions are monotonous). Finally, by Def. 6, we get the claim.

(Claim 22.2). Clearly, $M_1(B) \subseteq M_2(B)$. Then $M_1^*(B) \subseteq M_2^*(B)$, then $\widehat{P_{M_1^*}}(B) \subseteq \widehat{P_{M_2^*}}(B)$, and finally the claim. □

LEMMA 23 (Monotonicity of Behavior Transformations). *The behavioral transformation operators defined in Fig. 21 are monotonous.*

PROOF. It is easy to see that any composition of monotonous operators is monotonous. Because all the definitions in Fig. 21 are compositions of monotonous operators (set union, set concatenation, projections $[.]_{qq'}$ and $[.]_q$, constant functions, set interleaving, *proj* and intersection), the claim follows. □

A.5 General Procrastination

DEFINITION 24. *An operator on behaviors $f : \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$ is called* procrastinable *for a memory model M if all of the following are satisfied for all $B, B_i \subset \mathcal{B}$:*

1. $f(B)^M \subseteq f(B^M)$
2. $\lambda_{seq}(f(B_1), B_2) \subseteq f(\lambda_{seq}(B_1, B_2))$
3. $\lambda_{seq}(B_1, f(B_2)) \subseteq f(\lambda_{seq}(B_1, B_2))$
4. $\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_{k-1}, f(B_k), B_{k+1}, \ldots, B_n) \subseteq f(\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n))$
5. $\lambda_{cond,r}(f(B_1), B_2) \subseteq f(\lambda_{cond,r}(B_1, B_2))$
6. $\lambda_{cond,r}(B_1, f(B_2)) \subseteq f(\lambda_{cond,r}(B_1, B_2))$
7. $\lambda_{while,r}(f(B)) \subseteq f(\lambda_{while,r}(B))$
8. $\lambda_{local,r,x}(f(B)) \subseteq f(\lambda_{local,r,x}(B))$
9. $\lambda_{share,L,x}(f(B)) \subseteq f(\lambda_{share,L,x}(B))$
10. $obs(f(B)) \subseteq obs(B)$

THEOREM 25 (Generalized Soundness). *Let $(s \to s')$ be a local transformation, and let $M \subseteq \mathcal{T}$ be a memory model. Then the following condition is sufficient to guarantee soundness: there exists a transformation f that is procrastinable on M such that $[\![s']\!]_M \subseteq f([\![s]\!]_M)$.*

PROOF. We show that under the conditions stated in the theorem, $[\![c[s']]\!]_M \subseteq f([\![c[s]]\!]_M)$ for all contexts c; this then implies the claim: $obs_M(c[s']) = obs([\![c[s']]\!]_M) \subseteq obs(f([\![c[s]]\!]_M)) \subseteq obs([\![c[s]]\!]_M) = obs_M(c[s])$.

To prove that $[\![c[s']]\!]_M \subseteq f([\![c[s]]\!]_M)$ for all c, we proceed by induction over the structure of c.

- If $c = [\,]$, then $[\![c[s']]\!]_M = [\![s']\!]_M \subseteq f([\![s]\!]_M) = f([\![c[s]]\!]_M)$.
- If $c = c';\ s''$, then

$$\begin{array}{lll} [\![c[s']]\!]_M & \overset{15}{=} & \lambda_{seq}([\![c'[s']]\!]_M, [\![s'']\!]_M)^M \\ & \overset{ind.hyp.,23,22.1}{\subseteq} & \lambda_{seq}(f([\![c'[s]]\!]_M), [\![s'']\!]_M)^M \\ & \subseteq & (f(\lambda_{seq}([\![c'[s]]\!]_M, [\![s'']\!]_M)))^M \\ & \subseteq & f(\lambda_{seq}([\![c'[s]]\!]_M, [\![s'']\!]_M)^M) \\ & \overset{15}{=} & f([\![c[s])]\!]_M \end{array}$$

- If $c = s'';c'$, then proceed symmetrically.
- If $c = \mathbf{if}\ r\ \mathbf{then}\ c'\ \mathbf{else}\ s''$, then

$$\begin{array}{lll} [\![c[s']]\!]_M & \overset{15}{=} & \lambda_{cond,r}([\![c'[s']]\!]_M, [\![s'']\!]_M)^M \\ & \overset{ind.hyp.,23,22.1}{\subseteq} & \lambda_{while,r}(f([\![c'[s]]\!]_M), [\![s'']\!]_M)^M \\ & \subseteq & (f(\lambda_{while,r}([\![c'[s]]\!]_M, [\![s'']\!]_M)))^M \\ & \subseteq & f(\lambda_{while,r}([\![c'[s]]\!]_M, [\![s'']\!]_M)^M) \\ & \overset{15}{=} & [\![c[s]]\!]_M \end{array}$$

- If $c = \mathbf{if}\ r\ \mathbf{then}\ s''\ \mathbf{else}\ c'$, then proceed symmetrically.
- If $c = \mathbf{while}\ r\ \mathbf{do}\ c'$, then

$$\begin{array}{lll} [\![c[s']]\!]_M & \overset{15}{=} & \lambda_{while,r}([\![c'[s']]\!]_M)^M \\ & \overset{ind.hyp.,23,22.1}{\subseteq} & \lambda_{while,r}([\![c'[s]]\!]_M)^M \\ & \subseteq & \lambda_{while,r}(f([\![c'[s]]\!]_M))^M \\ & \subseteq & (f(\lambda_{while,r}([\![c'[s]]\!]_M)))^M \\ & \subseteq & f(\lambda_{while,r}([\![c'[s]]\!]_M)^M) \\ & \overset{15}{=} & f([\![c[s]]\!]_M) \end{array}$$

- If $c = \mathbf{local}\ r = x\ \mathbf{in}\ c'$ or $c = \mathbf{share}\ r = x\ \mathbf{in}\ c'$, then proceed analogously to previous case.
- If $c = s_1 \parallel \cdots \parallel c' \parallel \ldots s_n$, then $[\![c[s']]\!]_M =$

$$\begin{array}{l} \overset{15}{=} \lambda_{par,F_1,\ldots,F_n}([\![s_1]\!]_M, \ldots, [\![c'[s']]\!]_M, \ldots, [\![s_n]\!]_M)^M \\ \subseteq \lambda_{par,F_1,\ldots,F_n}([\![s_1]\!]_M, \ldots, f([\![c'[s]]\!]_M), \ldots, [\![s_n]\!]_M)^M \\ \subseteq (f(\lambda_{par,F_1,\ldots,F_n}([\![s_1]\!]_M, \ldots, [\![c'[s]]\!]_M, \ldots, [\![s_n]\!]_M)))^M \\ \subseteq f(\lambda_{par,F_1,\ldots,F_n}([\![s_1]\!]_M, \ldots, [\![c'[s]]\!]_M, \ldots, [\![s_n]\!]_M)^M) \\ \overset{15}{=} f([\![c[s]]\!]_M) \end{array}$$

□

A.6 Proof of Thm. 8

The following lemma captures the key property. Using this lemma, we can easily prove Theorem 8, by Definition 7 and Lemma 14.

LEMMA 26 (Monotonocity of Contexts). *Let $M \subseteq \mathcal{T}$ be a memory model, and let $s, s' \in \mathcal{S}$ be two snippets such that $[\![s']\!]_M \subseteq [\![s]\!]_M$. Then, for any context c, we have $[\![c[s']]\!]_M \subseteq [\![c[s]]\!]_M$.*

PROOF. Follows from Thm. 25, with f being the identity function which is procrastinable, as it obviously satsifies all the conditions in Def. 24. □

A.7 Auxiliary Lemmas

LEMMA 27. *Let $m \in \mathcal{T}$ be a simple rewrite rule, let $d \in \mathcal{T}$ be an arbitrary rewrite rule, and let $A_i \subset Evt^*$ for $i \in I$ with I some arbitrary index set. Then*

1. $\bigcup_{i\in I} m(A_i) = m(\bigcup_{i\in I} A_i)$
2. $\bigcup_{i\in I} d(A_i) \subseteq d(\bigcup_{i\in I} A_i)$

PROOF. (Claim 27.2). Because d is monotonous, we can apply Lemma 17.

(Claim 27.1). One direction follows by 27.2. For the other, assume $w \in m(\bigcup_{i\in I} A_i)$. Because m is simple, there exist $u, v, w_1, w_2 \in Evt^*$ such that $(\{u\}, \{v\}) \in G_m$ and $w = w_1vw_2$ and $w_1uw_2 \in \bigcup_{i\in I} A_i$. But that means there exists an i such that $w_1uw_2 \in A_i$ which implies $w \in m(A_i)$ and thus $w \in \bigcup_{i\in I} m(A_i)$. □

LEMMA 28 (Fixpoint for M). *Let $M \subseteq \mathcal{T}$ be a memory model, let $A \subseteq Evt^*$, let $\hat{A} \subseteq Evt^\infty$, and let $B \subseteq \mathcal{B}$. Then*

1. $M^*(M^*(A)) \subseteq M^*(A)$
2. $M^\infty(M^\infty(\hat{A})) \subseteq M^\infty(\hat{A})$
3. $(B^M)^M \subseteq B^M$

PROOF.

(Claim 28.1). Let $w \in M^*(M^*(A))$. By definition, this means $w \in \bigcup_{k\geq 0} M^k(M^*(A))$. Therefore, $w \in M^k(M^*(A))$ for some k. We now show that this implies $w \in M^*(A)$, by induction over k. If $k = 0$, then $w \in M^0(M^*(A)) = M^*(A)$. If $k > 0$, then $w \in M^k(M^*(A)) = M(M^{k-1}(M^*(A))$. By induction hypothesis and monotonicity of M, this implies $w \in M(M^*(A))$. By definition of the operator M, this implies that either $w \in M^*(A)$, or $w \in m(M^*(A))$ for some $m \in M$. In either case, because M is a memory model and its rewrite rules are thus simple, there must exist a u such that $m \in M(\{u\})$ and $u \in M^*(A)$. The latter implies that $u \in M^i(A)$ for some i. Therefore, $m \in M(\{u\}) \subseteq M(M^i(A)) \subseteq M^*(A)$.

(Claim 28.2). Let $w \in M^\infty(M^\infty(\hat{A}))$. By definition, this means that $w \in \bigcup_{k\geq 0}(\widehat{P_{M^*}})^k(M^\infty(\hat{A}))$. Therefore, there exists a k such that $w \in (\widehat{P_{M^*}})^k(M^\infty(\hat{A}))$. We now show that this implies $w \in M^\infty(\hat{A})$, by induction over k. For $k = 0$, we have $w \in (\widehat{P_{M^*}})^0(M^\infty(\hat{A})) = M^\infty(\hat{A})$. For $k > 0$, we have $w \in (\widehat{P_{M^*}})^k(M^\infty(\hat{A})) = w \in \widehat{P_{M^*}}((\widehat{P_{M^*}})^{k-1}(M^\infty(\hat{A})))$. By induction and monotonicity of $\widehat{P_{M^*}}$, this implies $w \in \widehat{P_{M^*}}(M^\infty(\hat{A}))$. By definition of M^∞, this means that there exists a decomposition $w = w_1 w_2 \cdots$ and sets $A_1, A_2, \cdots \subseteq Evt^*$ such that $w_i \in M^*(A_i)$ and $A_1 A_2 \cdots \subseteq M^\infty(\hat{A})$. Because M is a memory model (and its rewrite rules are simple), we further know that there exist elements $a_i \in A_i$ such that $w_i \in M^*(\{a_i\})$. Now, let $a = a_1 a_2 \cdots$. Then $a \in M^\infty(\hat{A})$, so there exists a $i \geq 0$ such that $a \in (\widehat{P_{M^*}})^i(\hat{A})$. But then

$$\begin{array}{rcl} w & = & w_1 w_2 \cdots \\ & \in & M^*(\{a_1\})M^*(\{a_2\})\cdots \\ & \subseteq & \widehat{P_{M^*}}(\{a_1\}\{a_2\}\cdots) \\ & = & \widehat{P_{M^*}}(\{a\}) \\ & \subseteq & \widehat{P_{M^*}}((\widehat{P_{M^*}})^i(\hat{A})) \\ & = & (\widehat{P_{M^*}})^{i+1}(\hat{A}) \\ & \subseteq & M^\infty(\hat{A}) \end{array}$$

(Claim 28.3). Directly from 28.1, 28.2 and Def. 6. □

A.8 Value Consistency

We now formally define value consistency. First, we define the set $CE(L, x, x')$ to be the set of all events that are consistent with location L having a pre-value of x and a post-value of x'. Then, we derive sets for value-consistent sequences. $Cons(L, x)$ is the set of all finite or infinite value-consistent sequences for location L and initial value x, and $Cons(L, x, x')$ is the set of all finite value-consistent sequences for location L, initial value x, and final value x'.

DEFINITION 29. *For $L \in \mathcal{L}$ and $x, x' \in \mathcal{X}$ we define the set $CE(L, x, x') \subseteq Evt$ by the following inference axioms with metavariables $L, L' \in \mathcal{L}$, $x, x', x'', x''' \in Val$:*

$$\langle ld_h\ L, x\rangle \in CE(L, x, x) \qquad \textit{(LOAD)}$$

$$\frac{L \neq L'}{\langle ld_h\ L, x\rangle \in CE(L', x', x')} \qquad \textit{(INDLOAD)}$$

$$\langle st_h\ L, x\rangle \in CE(L, x', x) \qquad \textit{(STORE)}$$

$$\frac{L \neq L'}{\langle st_h\ L, x\rangle \in CE(L', x', x')} \qquad \textit{(INDSTORE)}$$

$$\langle ldst_h\ L, x, x'\rangle \in CE(L, x, x') \qquad \textit{(LOADSTORE)}$$

$$\frac{L \neq L'}{\langle ldst_h\ L, x, x'\rangle \in CE(L', x'', x'')} \qquad \textit{(INDLOADSTORE)}$$

$$\langle mfence\rangle \in CE(L, x, x) \qquad \textit{(FENCE)}$$

$$\langle get\ x\rangle \in CE(L, x', x') \qquad \textit{(GET)}$$

$$\langle print\ x\rangle \in CE(L, x', x') \qquad \textit{(PRINT)}$$

DEFINITION 30. *For $L \in \mathcal{L}$ and $x, x' \in \mathcal{X}$ we define the sets $Cons(L, x, x') \subseteq Evt^*$, and $ConsL, x \subseteq Evt^\infty$ as follows:*

$$\begin{array}{l} Cons(L, x, x') = \{w \mid \exists x_0, \ldots x_{|w|} \in \mathcal{X} : \quad x = x_0, x' = x_n \\ \qquad \textit{and } \forall i \in \{1, \ldots, |w|\} : w[i] \in CE(L, x_{i-1}, x_i)\} \end{array}$$

$$\begin{array}{l} Cons(L, x) = \{w \in Evt^* \mid \exists x' \in \mathcal{X} : w \in Cons(L, x, x')\} \\ \qquad \cup \{\hat{w} \in Evt^\omega \mid \exists x_0, x_1, \cdots \in \mathcal{X} : \quad x = x_0 \\ \qquad\qquad \textit{and } \forall i \geq 1 : w[i] \in CE(L, x_{i-1}, x_i)\} \end{array}$$

LEMMA 31. *The following hold:*

1. *Let $n \geq 1$, let $a_1, a_2, \ldots a_n \in Evt^*$, and let $a = a_1 \cdots a_n$. Then $a \in Cons(L, x)$ if and only if there exist $x_0, x_1, \ldots, x_n \in \mathcal{X}$ such that $x = x_0$ and such that $a_i \in Cons(L, x_{i-1}, x_i)$ for $1 \leq i \leq n$.*
2. *Let $a_1, a_2, \cdots \in Evt^*$, and let $\hat{a} = a_1 a_2 \cdots$. Then $\hat{a} \in Cons(L, x)$ if and only if there exist $x_0, x_1, \cdots \in \mathcal{X}$ such that $x = x_0$ and $a_i \in Cons(L, x_{i-1}, x_i)$ for $i \geq 1$.*

PROOF. (Claim 31.1). ($\Rightarrow$) Because $a \in Cons(L, x)$ we know $a \in Cons(L, x, x')$ for some x'. That in turn means there exist values x_i, which we can use to prove $a_i \in Cons(L, x_{k_i}, x'_{k_i})$ for suitable $k_0 \leq k_1 \leq k_n \subset \{0, 1, \ldots, |a|\}$. ($\Leftarrow$) We can combine the value sequences that must exist because of $a_i \in Cons(L, x_{i-1}, x_i)$ into a single sequence that proves $a \in Cons(L, x, x')$, and thus $a \in Cons(L, x)$.

(Claim 31.2). Analogous to 31.1. □

A.9 Specialized Procrastination Lemmas

LEMMA 32. *Let $D \subseteq \mathcal{T}$. Then the following are true for all $A_1, A_2, \cdots \subseteq Evt^*$ and $\hat{A}_2 \subseteq Evt^\infty$:*

1. $P_D(A_1)P_D(A_2) \subseteq P_D(A_1 A_2)$
2. $P_D(A_1)\widehat{P_D}(\hat{A}_2) \subseteq \widehat{P_D}(A_1 \hat{A}_2)$
3. $P_D(A_1)P_D(A_2)\cdots \subseteq \widehat{P_D}(A_1 A_2 \ldots)$

PROOF. (Claim 32.1). Let $w \in P_D(A_1)P_D(A_2)$. Then there exist sets $A_1^1, A_1^2, \ldots, A_1^{n_1} \subseteq Evt^*$ and $A_2^1, A_2^2, \ldots, A_2^{n_2} \subseteq Evt^*$ such that we have $A_1^1 \cdots A_1^{n_1} \subseteq A_1$ and $A_2^1 \cdots A_2^{n_2} \subseteq A_2$, and such that $w \in D(A_1^1) \cdots D(A_1^{n_1})D(A_2^1) \cdots D(A_2^{n_2})$. But this implies $w \in \widehat{P_D}(A_1 A_2)$ because $A_1^1 \cdots A_1^{n_1} A_2^1 \cdots A_2^{n_2} \subseteq A_1 A_2$.

(Claim 32.2). (Claim 32.3). Analogously. □

LEMMA 33. *Let $D \subseteq \mathcal{T}$ satsify properties* 10.1 *and* 10.3*. Then the following are true for all $A \subseteq Evt^*$ and $\hat{A} \subseteq Evt^\infty$:*

1. $M^*(P_D(A)) \subseteq P_D(M^*(A))$
2. $M^\infty(\widehat{P_D}(\hat{A})) \subseteq \widehat{P_D}(M^\infty(\hat{A}))$

PROOF. (Claim 33.1). Let $w \in M^*(P_D(A))$. By Def. 6.2, this means $w \in \bigcup_{q\geq 0} M^k(P_D(A))$. Therefore $w \in M^k(P_D(A))$ for some k. We show that this implies $w \in P_D(M^*(A))$, by induction on k. If $k = 0$, then $w \in P_D(A) \subseteq P_D(M^*(A))$. If $k > 0$, then $w \in M(M^{k-1}(P_D(A)))$, and by the induction hypothesis (and because M is monotonous) we get $w \in M(P_D(M^*(A)))$. By definition, this means $w \in P_D(M^*(A)) \cup \bigcup_{m\in M} m(P_D(M^*(A)))$. If $w \in PDf(M^*(A))$, we are done. Otherwise, $w \in m(P_D(M^*(A)))$ for some $m \in M$. Now, by property 10.1 we know $m(P_D(M^*(A))) \subseteq P_D(M^*(M^*(A)))$, and by Lemma 28 we know $M^*(M^*(A)) \subseteq M^*(A)$. Thus (using monotonicity of P_D) we know $m \in P_D(M^*(A))$.

(Claim 33.2). Let $w \in M^\infty(\widehat{P_D}(\hat{A}))$. Then by definition, $w \in \bigcup_{k\geq 0}(\widehat{P_{M^*}})^k(\widehat{P_D}(\hat{A}))$. Therefore $w \in (\widehat{P_{M^*}})^k(\hat{A})$ for some k. We show that this implies $w \in \widehat{P_D}(M^\infty(\hat{A}))$, by induction on k. If $k = 0$, then $w \in \widehat{P_D}(\hat{A}) \subseteq \widehat{P_D}(M^\infty(\hat{A}))$. If $k > 0$, then $w \in \widehat{P_{M^*}}((\widehat{P_{M^*}})^{k-1}(\widehat{P_D}(\hat{A})))$. By induction and monotonocity, we get $w \in \widehat{P_{M^*}}(\widehat{P_D}(M^\infty(\hat{A})))$. Because M consists of simple transformations only, the effect of M^* is elementwise, implying that there exists a decomposition $w = w_1 w_2 \cdots$ and a sequence $c_1, c_2, \ldots$ (where $w_i, c_i \in Evt^*$) such that $w_i \in M^*(\{c_i\})$ and $c_1 c_2 \cdots \in (\widehat{P_D}(M^\infty(\hat{A})))$. The latter implies that $c_1 c_2 \cdots \in B_1 B_2 \cdots$ for some $A_i, B_i \subseteq Evt^*$ satisfying $B_i = D(A_i)$ and $A_1 A_2 \cdots \subseteq M^\infty(\hat{A})$. Specifically, $c_1 c_2 \cdots = b_1 b_2 \cdots$ for some b_i such that $b_i \in B_i$. Now, by property 10.3, we can assume without loss of generality that all b_i have length 0 or 1 (by choosing the decomposition $A_1 A_2 \cdots$ fine enough). Therefore, there exists an index sequence $0 = k_0 \leq k_1 \leq k_2 \cdots$ such that $c_i = b_{k_{i-1}} b_{k_{i-1}+1} \cdots b_{k_i}$. Now, let $A'_i = A_{k_{i-1}} A_{k_{i-1}+1} \cdots A_{k_i}$, and let $B'_i = B_{k_{i-1}} B_{k_{i-1}+1} \cdots B_{k_i}$. Then

$$\begin{array}{rcl}
w & \in & M^*(\{c_1\})M^*(\{c_2\})\cdots \\
& \subseteq & M^*(B'_1)M^*(B'_2)\cdots \\
& \subseteq & M^*(P_D(A'_1))M^*(P_D(A'_2))\cdots \\
& \overset{33.1}{\subseteq} & P_D(M^*(A'_1))P_D(M^*(A'_2))\cdots \\
& \overset{32.3}{\subseteq} & \widehat{P_D}(M^*(A'_1)M^*(A'_2)\cdots) \\
& \subseteq & \widehat{P_D}(\widehat{P_{M^*}}(A'_1A'_2\cdots)) \\
& \subseteq & \widehat{P_D}(M^\infty(A'_1A'_2\cdots)) \\
& \subseteq & \widehat{P_D}(M^\infty(M^\infty(\hat{A}))) \\
& \overset{\text{Lemma 28}}{\subseteq} & \widehat{P_D}(M^\infty(\hat{A}))
\end{array}$$

□

LEMMA 34. *Let $D \subseteq \mathcal{T}$ satisfy property 10.2. Then the following are true for all $A_1, A_2 \subseteq Evt^*$ and $\hat{A}, \hat{A}_1, \hat{A}_2 \subseteq Evt^\infty$:*

1. $P_D(A_1) \# P_D(A_2) \subseteq P_D(A_1 \# A_2)$
2. $P_D(A_1) \# \widehat{P_D}(\hat{A}_2) \subseteq \widehat{P_D}(A_1 \# \hat{A}_2)$
3. $\widehat{P_D}(\hat{A}_1) \# \widehat{P_D}(\hat{A}_2) \subseteq \widehat{P_D}(\hat{A}_1 \# \hat{A}_2)$

PROOF. (Claim 34.1). Let $w \in P_D(A_1) \# P_D(A_2)$. Then there exist $w_1 \in P_D(A_1)$ and $w_2 \in P_D(A_2)$ such that $w = w_1 w_2$. Therefore, we can find decompositions $A^1 \cdots A^k \subseteq A_1$ and $A^{k+1} \cdots A^n \subset A_2$ and $w_1 = w^1 \cdots w^k$ and $w_2 = w^{k+1} \cdots w^n$ such that $w^i \in D(A^i)$.Without loss of generality, we can assume that $|w^i| \leq 1$ (because of 10.2, and using singleton single-event sets to break down pieces where D does not modify the sequence). But this implies that $w = w^{\pi(1)} \cdots w^{\pi(n)}$ for some shuffle π of $\{1, \ldots, k\}$ and $\{k+1, \ldots, n\}$. This implies

$$w \in D(A^{\pi(1)}) \cdots D(A^{\pi(n)}) \subseteq P_D(A^{\pi(1)} \cdots A^{\pi(n)})$$

$$\subseteq P_D((A^1 \cdots A^k) \#(A^{k+1} \cdots A^n)) \subseteq P_D(A_1 \# A_2)$$

.

(Claim 34.2). (Claim 34.3). Analogously. □

LEMMA 35. *Let $D \subseteq \mathcal{T}$ satisfy property 10.3. Then the following are true for all $A \subseteq Evt^*$, $\hat{A} \subseteq Evt^\infty$, $L \in \mathcal{L}$ and $x \in \mathcal{X}$:*

1. $proj_{-L}(P_D(A) \cap Cons(L, x)) \subseteq P_D(proj_{-L}(A \cap Cons(L, x)))$
2. $proj_{-L}(\widehat{P_D}(\hat{A}) \cap Cons(L, x)) \subseteq \widehat{P_D}(proj_{-L}(\hat{A} \cap Cons(L, x)))$

PROOF. (Claim 35.1). Let $w \in proj_{-L}(P_D(A) \cap Cons(L, x))$. By definition, this means that there exist sequences $b_1, \ldots, b_n \subseteq Evt^*$ and sets $A_1, \ldots, A_n \subseteq Evt^*$ such that $w = proj_{-L}(b_1 \cdots b_n)$ and $b_1 \cdots b_n \in Cons(L, x)$ and $b_i \in D(A_i)$ and $A_1 \cdots A_n \subseteq A$. Now, by Lemma 31, this implies that there exist $x_0, \ldots, x_n \in \mathcal{X}$ such that $x_0 = x$ and $b_i \in Cons(L, x_{i-1}, x_i)$. By property 10.3, this implies that there exist $a_i \in A_i$ such that $a_i \in Cons(L, x_{i-1}, x_i)$ (and thus $a_1 \cdots a_n \in Cons(L, x)$ by Lemma 31) and such that $proj_{-L}(\{b_i\}) \subseteq D(proj_{-L}(\{a_i\}))$. Now,

$$\begin{array}{rcl}
w & = & proj_{-L}(b_1 \cdots b_n) \\
& \in & proj_{-L}(\{b_1\}) \cdots proj_{-L}(\{b_n\}) \\
& \subseteq & D(proj_{-L}(\{a_1\})) \cdots D(proj_{-L}(\{a_n\})) \\
& \subseteq & P_D(proj_{-L}(\{a_1\}) \cdots proj_{-L}(\{a_n\}))) \\
& = & P_D(proj_{-L}(\{a_1 \cdots a_n\})) \\
& = & P_D(proj_{-L}(\{a_1 \cdots a_n\} \cap Cons(L, x)) \\
& \subseteq & P_D(proj_{-L}(A \cap Cons(L, x))
\end{array}$$

(Claim 35.2). Let $w \in proj_{-L}(\widehat{P_D}(\hat{A}) \cap Cons(L, x))$. By definition, this means that there exist sequences $b_1, b_2, \cdots \subseteq Evt^*$ and sets $A_1, A_2, \cdots \subseteq Evt^*$ such that $w = proj_{-L}(b_1 b_2 \cdots)$ and $b_1 b_2 \cdots \in Cons(L, x)$ and $b_i \in D(A_i)$ and $A_1 A_2 \cdots \subseteq \hat{A}$. Now, by Lemma 31, this implies that there exist $x_0, x_1, \cdots \in \mathcal{X}$ such that $x_0 = x$ and $b_i \in Cons(L, x_{i-1}, x_i))$. By property 10.3, this implies that there exist $a_i \in A_i$ such that $a_i \in Cons(L, x_{i-1}, x_i)$ (and thus $a_1 a_2 \cdots \in Cons(L, x)$ by Lemma 31) and such that $proj_{-L}(\{b_i\}) \subseteq D(proj_{-L}(\{a_i\}))$. Now,

$$\begin{array}{rcl}
w & = & proj_{-L}(b_1 b_2 \cdots) \\
& \in & proj_{-L}(\{b_1\}) proj_{-L}(\{b_2\}) \cdots \\
& \subseteq & D(proj_{-L}(\{a_1\})) D(proj_{-L}(\{a_2\})) \cdots \\
& \subseteq & \widehat{P_D}(proj_{-L}(\{a_1\}) proj_{-L}(\{a_2\})) \cdots) \\
& = & \widehat{P_D}(proj_{-L}(\{a_1 a_2 \cdots\})) \\
& = & \widehat{P_D}(proj_{-L}(\{a_1 a_2 \cdots\} \cap Cons(L, x)) \\
& \subseteq & \widehat{P_D}(proj_{-L}(\hat{A} \cap Cons(L, x))
\end{array}$$

□

LEMMA 36. *Let $D \subseteq \mathcal{T}$ satisfy property 10.4. Then the following are true for all $A \subseteq Evt^*$ and $\hat{A} \subseteq Evt^\infty$:*

1. $obs(P_D(A)) \subseteq obs(A)$
2. $obs(\widehat{P_D}(\hat{A})) \subseteq obs(\hat{A})$

PROOF. (Claim 36.1). Let $o \in obs(P_D(A))$. Then there exist $A_1, \ldots, A_n$ and $b_1, \ldots, b_n$ such that $A_1 \cdots A_n \subseteq A$ and $b_i \in D(A_i)$ and $o = proj_{Ext}(b_1 \cdots b_n)$. By 10.4, we know

$proj_{Ext}(b_i) \in proj_{Ext}(A_i)$. Therefore,

$$\begin{array}{rcl} o & = & proj_{Ext}(b_1 \cdots b_n) \\ & = & proj_{Ext}(b_1) \cdots proj_{Ext}(b_n) \\ & \in & proj_{Ext}(A_1) \cdots proj_{Ext}(A_n) \\ & = & proj_{Ext}(A_1 \cdots A_n) \\ & \subseteq & proj_{Ext}(A) \\ & = & obs(A). \end{array}$$

(Claim 36.1). Let $o \in obs(\widehat{P_D}(A))$. Then there exist $A_1, A_2, \ldots$ and $b_1, b_2, \ldots$ such that $A_1 A_2 \cdots \subseteq \hat{A}$ and $b_i \in D(A_i)$ and $o = \nabla proj_{Ext}(b_1 b_2 \cdots)$. By 10.4, we know $proj_{Ext}(b_i) \in proj_{Ext}(A_i)$. Therefore,

$$\begin{array}{rcl} o & = & \nabla proj_{Ext}(b_1 b_2 \cdots) \\ & = & \nabla proj_{Ext}(b_1) \cdots proj_{Ext}(b_n) \\ & \in & \nabla (proj_{Ext}(A_1) proj_{Ext}(A_2) \cdots) \\ & = & \nabla proj_{Ext}(A_1 \cdots A_n) \\ & \subseteq & \nabla proj_{Ext}(\hat{A}) \\ & = & obs(\hat{A}). \end{array}$$

□

A.10 Parallel Rewriting of Behaviors

DEFINITION 37. *Let $B \subseteq \mathcal{B}$, and let $D \subset \mathcal{T}$. Then we define $f_D : \mathcal{P}(\mathcal{B}) \to \mathcal{P}(\mathcal{B})$ by*

$$[f_D(B)]_{qq'} = P_D([B]_{qq'})$$

$$[f_D(B)]_q = \widehat{P_D}([B]_q)$$

LEMMA 38. *Let $B \subseteq \mathcal{B}$, and let $D \subset \mathcal{T}$. Then f_D is monotonous and expanding.*

PROOF. Follows from Lemmas 19 and 21. □

A.11 Proof of Thm. 11

To prove Thm. 11, we use the following theorem which states that f_D is procrastinable if D is a procrastinable subset. With f_D procrastinable, it is easy to see that $(f_D)^k$ is also procrastinable for all $k \geq 0$ (we can prove by induction to each individual claim of Def. 24 that if f is procrastinable, so is f^k). This means that we can apply Thm. 25 to conclude the proof of Thm. 11, because the conditions stated in the latter are equivalent to $[\![s']\!]_M \subseteq (f_D)^k([\![s]\!]_M)$ (by Def. 37).

THEOREM 39. *If D is procrastinable of M (in the sense of Def. 10), then f_D is procrastinable on M (in the sense of Def. 24).*

PROOF. We show that f_D satisfies properties 1 through 10 of Def. 24;

All proof cases are shown in Fig. 22 and Fig. 23. Cases 1 through 9 are split into finite behaviors $[]_{qq'}$ and infinite behaviors $[]_q$. Case 10 is split into two directions. For simplicity, we omit index sets and collapse definitions where the details are not relevant for the proof structure.

We annotate the important Lemmas or Definitions we use; however, we pervasively use the following fact without annotating the monotonicity of all operators involved: if we enlarge subexpressions (that is, replace subexpression S by some S' such that $S \subseteq S'$), then the whole expression enlarges, under the condition that it depends on the subexpression in a monotonous way. □

A.12 Proof of Thm. 12

Property 2 is immediate. Property 3 is straightforward. Property 4 is trivial as none of the rewrite rules involves any external events. Property 1 is where we need to work.

The general procedure for a memory model M and set of rules D is to examine the left-hand side of each rule $m \in M$, trying to overlap it with the right-hand sides of non-trivial parallel applications of D. If no such overlap scenario is possible, commutation ($m(P_D(A)) = P_D(m(A))$) holds, implying property 10.1. If it is possible, we need to show for each overlap scenario that the image of $m(P_D(A))$ is contained in $P_D(M^*(A))$, that is, each element of the image can be obtained by first applying M zero or more times, and then (optionally) applying D parallely.

- $\boxed{M = SC = \{\}, \text{ and } D \text{ arbitrary}}$

  Because $SC = \emptyset$, it is immediate that any D satisfies property 10.1.

- $\boxed{M = X86 = \{\textbf{srsrl}, \textbf{ersrlf}\}, \text{ and } D = \{\textbf{edl}_R\}}$

  $\boxed{m = \textbf{srsrl}}$ The only overlap scenario is of the form

  $$\begin{array}{rcl} \langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle\langle ld_R\ L', x'\rangle & \in & A \\ \langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle & \in & P_D(A) \\ \langle ld_R\ L', x'\rangle\langle st_R\ L, x\rangle & \in & \textbf{srsrl}(P_D(A)) \end{array}$$

  with image in $P_D(M^*(A))$:

  $$\begin{array}{rcl} \langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle\langle ld_R\ L', x'\rangle & \in & A \\ \langle ld_R\ L', x'\rangle\langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle & \in & \textbf{srsrl}(A) \\ \langle ld_R\ L', x'\rangle\langle ld_R\ L', x'\rangle\langle st_R\ L, x\rangle & \in & \textbf{srsrl}(\textbf{srsrl}(A)) \\ \langle ld_R\ L', x'\rangle\langle st_R\ L, x\rangle & \in & P_D(\textbf{srsrl}(\textbf{srsrl}(A))) \end{array}$$

  $\boxed{m = \textbf{ersrlf}}$ The only overlap scenario is of the form

  $$\begin{array}{rcl} \langle st_R\ L, x\rangle\langle ld_R\ L, x\rangle\langle ld_R\ L, x\rangle & \in & A \\ \langle st_R\ L, x\rangle\langle ld_R\ L, x\rangle & \in & P_D(A) \\ \langle st_R\ L, x\rangle & \in & \textbf{ersrlf}(P_D(A)) \end{array}$$

  with image in $M^*(A)$ (and thus in $P_D(M^*(A))$):

  $$\begin{array}{rcl} \langle st_R\ L, x\rangle\langle ld_R\ L, x\rangle\langle ld_R\ L, x\rangle & \in & A \\ \langle st_R\ L, x\rangle\langle ld_R\ L, x\rangle & \in & \textbf{ersrlf}(A) \\ \langle st_R\ L, x\rangle & \in & \textbf{ersrlf}(\textbf{ersrlf}(A)) \end{array}$$

- $\boxed{M = X86 = \{\textbf{srsrl}, \textbf{ersrlf}\}, \text{ and } D = \{\textbf{eds}_R\}}$

  $\boxed{m = \textbf{srsrl}}$ The only overlap scenario is of the form

  $$\begin{array}{rcl} \langle st_R\ L, x\rangle\langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle & \in & A \\ \langle st_R\ L, x\rangle\langle ld_R\ L', x'\rangle & \in & P_D(A) \\ \langle ld_R\ L', x'\rangle\langle st_R\ L, x\rangle & \in & \textbf{srsrl}(P_D(A)) \end{array}$$

1.

$\boxed{\left[f_D(B)^M\right]_{qq'}}$

$\overset{6,\,37}{=} M^*(P_D([B]_{qq'}))$
$\overset{33.1}{\subseteq} P_D(M^*([B]_{qq'}))$
$\overset{37,\,6}{=} \left[f_D(B^M)\right]_{qq'}$

$\boxed{\left[f_D(B)^M\right]_{q}}$

$\overset{6,\,37}{=} M^\infty(\widehat{P_D}([B]_q))$
$\overset{33.2}{\subseteq} \widehat{P_D}(M^\infty([B]_q))$
$\overset{37,\,6}{=} \left[f_D(B^M)\right]_{q}$

2.

$\boxed{[\lambda_{seq}(f_D(B_1), B_2)]_{qq'}}$

$\overset{\text{Fig. 21}}{=} \bigcup_p [f_D(B_1)]_{qp} [B_2]_{pq'}$
$\overset{37}{=} \bigcup_p P_D([B_1]_{qp}) [B_2]_{pq'}$
$\overset{32.1,\,21}{\subseteq} \bigcup_p P_D([B_1]_{qp} [B_2]_{pq'})$
$\overset{19,\,17}{\subseteq} P_D\left(\bigcup_p [B_1]_{qp} [B_2]_{pq'}\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{seq}(B_1, B_2)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{seq}(B_1, B_2))]_{qq'}$

$\boxed{[\lambda_{seq}(f_D(B_1), B_2)]_{q}}$

$\overset{\text{Fig. 21}}{=} [f_D(B_1)]_q \cup \bigcup_p [f_D(B_1)]_{qp} [B_2]_p$
$\overset{37}{=} \widehat{P_D}([B_1]_q) \cup \bigcup_p P_D([B_1]_{qp}) [B_2]_p$
$\overset{32.2,\,21}{\subseteq} \widehat{P_D}([B_1]_q) \cup \bigcup_p \widehat{P_D}([B_1]_{qp} [B_2]_p)$
$\overset{19,\,17}{\subseteq} \widehat{P_D}\left([B_1]_q \cup \bigcup_p [B_1]_{qp} [B_2]_p\right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{seq}(B_1, B_2)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{seq}(B_1, B_2))]_q$

3.

$\boxed{[\lambda_{seq}(B_1, f_D(B_2))]_{qq'}}$

$\overset{\text{Fig. 21}}{=} \bigcup_p [B_1]_{qp} [f_D(B_2)]_{pq'}$
$\overset{37}{=} \bigcup_p [B_1]_{qp} P_D([B_2]_{pq'})$
$\overset{32.1,\,21}{\subseteq} \bigcup_p P_D([B_1]_{qp} [B_2]_{pq'})$
$\overset{19,\,17}{\subseteq} P_D\left(\bigcup_p [B_1]_{qp} [B_2]_{pq'}\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{seq}(B_1, B_2)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{seq}(B_1, B_2))]_{qq'}$

$\boxed{[\lambda_{seq}(B_1, f_D(B_2))]_{q}}$

$\overset{\text{Fig. 21}}{=} [B_1]_q \cup \bigcup_p [B_1]_{qp} [f_D(B_2)]_p$
$\overset{37}{=} [B_1]_q \cup \bigcup_p [B_1]_{qp} \widehat{P_D}([B_2]_p)$
$\overset{32.2,\,21}{\subseteq} \widehat{P_D}([B_1]_q) \cup \bigcup_p \widehat{P_D}([B_1]_{qp} [B_2]_p)$
$\overset{19,\,17}{\subseteq} \widehat{P_D}\left([B_1]_q \cup \bigcup_p [B_1]_{qp} [B_2]_p\right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{seq}(B_1, B_2)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{seq}(B_1, B_2))]_q$

4.

$\boxed{[\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, f_D(B_k), \ldots, B_n)]_{qq'} \text{ where not } \left(\forall r \in \bigcup_k F_k : q'(r) = q(r)\right)}$

$\overset{\text{Fig. 21}}{=} \emptyset$
$= P_D(\emptyset)$
$\overset{\text{Fig. 21}}{=} P_D([\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n)]_{qq'})$
$\overset{37}{=} [f_D(\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n))]_{qq'}$

$\boxed{[\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, f_D(B_k), \ldots, B_n)]_{qq'} \text{ where } \left(\forall r \in \bigcup_k F_k : q'(r) = q(r)\right)}$

$\overset{38,\,23}{\subseteq} [\lambda_{par,F_1,\ldots,F_n}(f_D(B_1), \ldots, f_D(B_n))]_{qq'}$
$\overset{\text{Fig. 21, 37}}{=} \#_k \bigcup_p P_D\left([B_k]_{qp}\right)$
$\overset{19,\,17}{=} \#_k P_D\left(\bigcup_p [B_k]_{qp}\right)$
$\overset{34.1}{=} P_D\left(\#_k \bigcup_p [B_k]_{qp}\right)$
$\overset{\text{Fig. 21}}{=} P_D([\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n)]_{qq'})$
$\overset{37}{=} [f_D(\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n))]_{qq'}$

$\boxed{[\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, f_D(B_k), \ldots, B_n)]_{q}}$

$\overset{38,\,23}{\subseteq} [\lambda_{par,F_1,\ldots,F_n}(f_D(B_1), \ldots, f_D(B_n))]_q$
$\overset{\text{Fig. 21, 37}}{=} \bigcup_D \left((\#_k \bigcup_p P_D([B_k]_{qp})) \,\#\, (\#_k \widehat{P_D}([B_k]_q)) \right)$
$\overset{19,\,17}{\subseteq} \bigcup_D \left((\#_k P_D\left(\bigcup_p [B_k]_{qp}\right)) \,\#\, (\#_k \widehat{P_D}([B_k]_q)) \right)$
$\overset{34.2,\,34.3}{\subseteq} \bigcup_D \widehat{P_D}\left((\#_k \bigcup_p [B_k]_{qp}) \,\#\, (\#_k [B_k]_q) \right)$
$\overset{19,\,17}{\subseteq} \widehat{P_D}\left(\bigcup_D \left((\#_k \bigcup_p [B_k]_{qp}) \,\#\, (\#_k [B_k]_q) \right) \right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{par,F_1,\ldots,F_n}(B_1, \ldots, B_n))]_q$

5.

$\boxed{[\lambda_{cond,r}(f_D(B_1), B_2)]_{qq'}}$

$\overset{38,\,23}{\subseteq} [\lambda_{cond,r}(f_D(B_1), f_D(B_2))]_{qq'}$
$\overset{\text{Fig. 21}}{=} [f_D(B_i)]_{qq'}$
$\overset{37}{=} P_D\left([B_i]_{qq'}\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{cond,r}(B_1, B_2)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{cond,r}(B_1, B_2))]_{qq'}$

$\boxed{[\lambda_{cond,r}(f_D(B_1), B_2)]_{q}}$

$\overset{38,\,23}{\subseteq} [\lambda_{cond,r}(f_D(B_1), f_D(B_2))]_q$
$\overset{\text{Fig. 21}}{=} [f_D(B_i)]_q$
$\overset{37}{=} \widehat{P_D}\left([B_i]_q\right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{cond,r}(B_1, B_2)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{cond,r}(B_1, B_2))]_q$

6.

$\boxed{[\lambda_{cond,r}(B_1, f_D(B_2))]_{qq'}}$

$\overset{38,\,23}{\subseteq} [\lambda_{cond,r}(f_D(B_1), f_D(B_2))]_{qq'}$
$\overset{\text{Fig. 21}}{=} [f_D(B_i)]_{qq'}$
$\overset{37}{=} P_D\left([B_i]_{qq'}\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{cond,r}(B_1, B_2)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{cond,r}(B_1, B_2))]_{qq'}$

$\boxed{[\lambda_{cond,r}(B_1, f_D(B_2))]_{q}}$

$\overset{38,\,23}{\subseteq} [\lambda_{cond,r}(f_D(B_1), f_D(B_2))]_q$
$\overset{\text{Fig. 21}}{=} [f_D(B_i)]_q$
$\overset{37}{=} \widehat{P_D}\left([B_i]_q\right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{cond,r}(B_1, B_2)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{cond,r}(B_1, B_2))]_q$

Figure 22. Cases for the proof of Thm. 39 (continued in Fig. 23).

7.

$\boxed{[\lambda_{\textit{while},r}(f_D(B))]_{qq'}}$

$\overset{\text{Fig. 21}}{=} \bigcup_{q_0,\ldots q_n} [f_D(B)]_{q_0q_1} \cdots [f_D(B)]_{q_{n-1}q_n}$
$\overset{37}{=} \bigcup_{q_0,\ldots q_n} P_D([B]_{q_0q_1}) \cdots P_D([B]_{q_{n-1}q_n})$
$\overset{32.1}{\subseteq} \bigcup_{q_0,\ldots q_n} P_D\left([B]_{q_0q_1} \cdots [B]_{q_{n-1}q_n}\right)$
$\overset{19,17}{\subseteq} P_D\left(\bigcup_{q_0,\ldots q_n} [B]_{q_0q_1} \cdots [B]_{q_{n-1}q_n}\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{\textit{while},r}(B)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{\textit{while},r}(B))]_{qq'}$

$\boxed{[\lambda_{\textit{while},r}(f_D(B))]_{q}}$

$\overset{\text{Fig. 21}}{=} \left(\bigcup_{q_0,q_1,\ldots} [f_D(B)]_{q_0q_1} [f_D(B)]_{q_1q_2} \cdots\right) \cup \left(\bigcup_{q_0,\ldots q_{n-1}} [f_D(B)]_{q_0q_1} \cdots [f_D(B)]_{q_{n-2}q_{n-1}} [f_D(B)]_{q_{n-1}}\right)$
$\overset{37}{=} \left(\bigcup_{q_0,q_1,\ldots} P_D([B]_{q_0q_1}) P_D([B]_{q_1q_2}) \cdots\right) \cup \left(\bigcup_{q_0,\ldots q_{n-1}} P_D([B]_{q_0q_1}) \cdots P_D([B]_{q_{n-2}q_{n-1}}) \widehat{P_D}([B]_{q_{n-1}})\right)$
$\overset{32.3,\,32.2}{\subseteq} \left(\bigcup_{q_0,q_1,\ldots} \widehat{P_D}\left([B]_{q_0q_1} [B]_{q_1q_2} \cdots\right)\right) \cup \left(\bigcup_{q_0,\ldots q_{n-1}} \widehat{P_D}\left([B]_{q_0q_1} \cdots [B]_{q_{n-2}q_{n-1}} [B]_{q_{n-1}}\right)\right)$
$\overset{19,17}{\subseteq} \widehat{P_D}\left(\left(\bigcup_{q_0,q_1,\ldots} [B]_{q_0q_1} [B]_{q_1q_2} \cdots\right) \cup \left(\bigcup_{q_0,\ldots q_{n-1}} \widehat{P_D}\, [B]_{q_0q_1} \cdots [B]_{q_{n-2}q_{n-1}} [B]_{q_{n-1}}\right)\right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}([\lambda_{\textit{while},r}(B)]_q)$
$\overset{37}{=} [f_D(\lambda_{\textit{while},r}(B))]_q$

8.

$\boxed{[\lambda_{\textit{local},r,x}(f_D(B))]_{qq'} \text{ where } q'(r) \neq q(r)}$

$\overset{\text{Fig. 21}}{=} \emptyset$
$= P_D(\emptyset)$
$\overset{\text{Fig. 21}}{=} P_D([\lambda_{\textit{local},r,x}(B)]_{qq'})$
$\overset{37}{=} [f_D(\lambda_{\textit{local},r,x}(B))]_{qq'}$

$\boxed{[\lambda_{\textit{local},r,x}(f_D(B))]_{qq'} \text{ where } q'(r) = q(r)}$

$\overset{\text{Fig. 21}}{=} \bigcup_{x'} [f_D(B)]_{pp'}$
$\overset{37}{=} \bigcup_{x'} P_D([B]_{pp'})$
$\overset{19,17}{\subseteq} P_D\left(\bigcup_{x'} [B]_{pp'}\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{\textit{local},r,x}(B)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{\textit{local},r,x}(B))]_{qq'}$

$\boxed{[\lambda_{\textit{local},r,x}(f_D(B))]_{q}}$

$\overset{\text{Fig. 21}}{=} [f_D(B)]_p$
$\overset{37}{=} \widehat{P_D}([B]_p)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{\textit{local},r,x}(B)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{\textit{local},r,x}(B))]_q$

9.

$\boxed{[\lambda_{\textit{share},L,x}(f_D(B))]_{qq'}}$

$\overset{\text{Fig. 21}}{=} \mathit{proj}_{-L}([f_D(B)]_{qq'} \cap \mathit{Cons}(L,x))$
$\overset{37}{=} \mathit{proj}_{-L}(P_D([B]_{qq'}) \cap \mathit{Cons}(L,x))$
$\overset{35.1}{\subseteq} P_D\left(\mathit{proj}_{-L}([B]_{qq'} \cap \mathit{Cons}(L,x))\right)$
$\overset{\text{Fig. 21}}{=} P_D\left([\lambda_{\textit{share},L,x}(B)]_{qq'}\right)$
$\overset{37}{=} [f_D(\lambda_{\textit{share},L,x}(B))]_{qq'}$

$\boxed{[\lambda_{\textit{share},L,x}(f_D(B))]_{q}}$

$\overset{\text{Fig. 21}}{=} \mathit{proj}_{-L}([f_D(B)]_q \cap \mathit{Cons}(L,x))$
$\overset{37}{=} \mathit{proj}_{-L}(\widehat{P_D}([B]_q) \cap \mathit{Cons}(L,x))$
$\overset{35.2}{\subseteq} \widehat{P_D}\left(\mathit{proj}_{-L}([B]_q \cap \mathit{Cons}(L,x))\right)$
$\overset{\text{Fig. 21}}{=} \widehat{P_D}\left([\lambda_{\textit{share},L,x}(B)]_q\right)$
$\overset{37}{=} [f_D(\lambda_{\textit{share},L,x}(B))]_q$

10.

$\boxed{\mathit{obs}(f_D(B))}$

$\overset{13}{=} \mathit{obs}(\bigcup_{qq'} [f_D(B)]_{qq'}) \cup \mathit{obs}(\bigcup_q [f_D(B)]_q)$
$\overset{37}{=} \mathit{obs}(\bigcup_{qq'} P_D([B]_{qq'})) \cup \mathit{obs}(\bigcup_q \widehat{P_D}([B]_q))$
$\overset{19,17}{\subseteq} \mathit{obs}(P_D(\bigcup_{qq'} [B]_{qq'})) \cup \mathit{obs}(\widehat{P_D}(\bigcup_q [B]_q))$
$\overset{36.1,\,36.2}{\subseteq} \mathit{obs}(\bigcup_{qq'} [B]_{qq'}) \cup \mathit{obs}(\bigcup_q [B]_q)$
$\overset{13}{=} \mathit{obs}(B)$

$\boxed{\mathit{obs}(B)}$

$\overset{13}{=} \mathit{obs}(\bigcup_{qq'} [B]_{qq'}) \cup \mathit{obs}(\bigcup_q [B]_q)$
$\overset{21}{\subseteq} \mathit{obs}(\bigcup_{qq'} P_D([B]_{qq'})) \cup \mathit{obs}(\bigcup_q \widehat{P_D}([B]_q))$
$\overset{37}{=} \mathit{obs}(\bigcup_{qq'} [f_D(B)]_{qq'})) \cup \mathit{obs}(\bigcup_q [f_D(B)]_q)$
$\overset{13}{=} \mathit{obs}(f_D(B))$

Figure 23. Cases for the proof of Thm. 39 (continued from Fig. 22).

with image in $P_D(M^*(A))$:

$$\begin{array}{rcl}\langle st_R\ L,x\rangle\langle st_R\ L,x\rangle\langle ld_R\ L',x'\rangle & \in & A\\ \langle st_R\ L,x\rangle\langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle & \in & \textbf{srsrl}(A)\\ \langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle\langle st_R\ L,x\rangle & \in & \textbf{srsrl}(\textbf{srsrl}(A))\\ \langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle & \in & P_D(\textbf{srsrl}(\textbf{srsrl}(A)))\end{array}$$

$\boxed{m = \textbf{ersrlf}}$ The only overlap scenario is of the form

$$\begin{array}{rcl}\langle st_R\ L,x\rangle\langle st_R\ L,x\rangle\langle ld_R\ L,x\rangle & \in & A\\ \langle st_R\ L,x\rangle\langle ld_R\ L,x\rangle & \in & P_D(A)\\ \langle st_R\ L,x\rangle & \in & \textbf{ersrlf}(P_D(A))\end{array}$$

with image in $P_D(M^*(A))$:

$$\begin{array}{rcl}\langle st_R\ L,x\rangle\langle st_R\ L,x\rangle\langle ld_R\ L,x\rangle & \in & A\\ \langle st_R\ L,x\rangle\langle st_R\ L,x\rangle & \in & \textbf{ersrlf}(A)\\ \langle st_R\ L,x\rangle & \in & P_D(\textbf{ersrlf}(A))\end{array}$$

- $\boxed{M = X86 = \{\textbf{srsrl}, \textbf{ersrlf}\}, \text{ and } D = \{\textbf{eil}_R\}}$

$\boxed{m = \textbf{srsrl}}$ The only overlap scenario is of the form

$$\begin{array}{rcl}\langle st_R\ L,x\rangle\{\langle ld_R\ L_1,*\rangle\}\cdots\\ \quad\{\langle ld_R\ L_n,*\rangle\}\langle ld_R\ L',x'\rangle & \subseteq & A\\ \langle st_R\ L,x\rangle\langle ld_R\ L',x'\rangle & \in & P_D(A)\\ \langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle & \in & \textbf{srsrl}(P_D(A))\end{array}$$

Now consider the following $w \in A$:

$$w = \langle st_R\ L,x\rangle\langle ld_R\ L_1,x_1\rangle\cdots\langle ld_R\ L_n,x_n\rangle\langle ld_R\ L',x'\rangle$$

where $x_i = x$ whenever $L = L_i$, or arbitrary otherwise. Then, by repeatedly applying **srsrl** or **ersrlf**, we can move the store all the way to the right, which proves

$$\langle ld_R\ L_{i_1},x_{i_1}\rangle\cdots\langle ld_R\ L_{i_n},x_{i_n}\rangle\langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle \in M^*(A)$$

where $i_1,\ldots,i_n$ is the subsequence of $1,\ldots,k$ consisting of all indices i for which $L_i \neq L$. Finally, because the x_{i_j} were arbitrary, this implies

$$\{\langle ld_R\ L_{i_1},*\rangle\cdots\langle ld_R\ L_{i_n},*\rangle\}\langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle \subseteq M^*(A)$$

and we can thus eliminate the loads, giving

$$\langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle \subseteq P_D(M^*(A)).$$

$\boxed{m = \textbf{ersrlf}}$ The only overlap scenario is of the form

$$\begin{array}{rcl}\langle st_R\ L,x\rangle\{\langle ld_R\ L_1,*\rangle\}\cdots\\ \quad\{\langle ld_R\ L_n,*\rangle\}\langle ld_R\ L,x\rangle & \subseteq & A\\ \langle st_R\ L,x\rangle\langle ld_R\ L,x\rangle & \in & P_D(A)\\ \langle st_R\ L,x\rangle & \in & \textbf{ersrlf}(P_D(A))\end{array}$$

Now consider again the following $w \in A$:

$$w = \langle st_R\ L,x\rangle\langle ld_R\ L_1,x_1\rangle\cdots\langle ld_R\ L_n,x_n\rangle\langle ld_R\ L,x\rangle$$

where $x_i = x$ whenever $L = L_i$, or arbitrary otherwise. Then, by repeatedly applying **srsrl** or **ersrlf**, we can move the store all the way to the right and forward the load on the right, which proves

$$\langle ld_R\ L_{i_1},x_{i_1}\rangle\cdots\langle ld_R\ L_{i_n},x_{i_n}\rangle\langle st_R\ L,x\rangle \in M^*(A)$$

where $i_1,\ldots,i_n$ is the subsequence of $1,\ldots,k$ consisting of all indices i for which $L_i \neq L$. Finally, because the x_{i_j} were arbitrary, this implies

$$\{\langle ld_R\ L_{i_1},*\rangle\cdots\langle ld_R\ L_{i_n},*\rangle\}\langle st_R\ L,x\rangle \subseteq M^*(A)$$

and we can thus eliminate the loads, giving

$$\langle st_R\ L,x\rangle \subseteq P_D(M^*(A))$$

- $\boxed{M = X86 = \{\textbf{srsrl}, \textbf{ersrlf}\}, \text{ and } D = \{\textbf{iil}_R\}}$

$\boxed{m = \textbf{srsrl}}$ The only overlap scenario is of the form

$$\begin{array}{rcl}\langle st_R\ L,x\rangle & \in & A\\ \langle st_R\ L,x\rangle\langle ld_R\ L',x'\rangle & \in & P_D(A)\\ \langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle & \in & \textbf{srsrl}(P_D(A))\end{array}$$

with image in $P_D(M^*(A))$:

$$\begin{array}{rcl}\langle st_R\ L,x\rangle & \in & A\\ \langle st_R\ L,x\rangle & \in & M^*(A)\\ \langle ld_R\ L',x'\rangle\langle st_R\ L,x\rangle & \in & P_D(M^*(A))\end{array}$$

$\boxed{m = \textbf{ersrlf}}$ The only overlap scenario is of the form

$$\begin{array}{rcl}\langle st_R\ L,x\rangle & \in & A\\ \langle st_R\ L,x\rangle\langle ld_R\ L,x\rangle & \in & P_D(A)\\ \langle st_R\ L,x\rangle & \in & \textbf{ersrlf}(P_D(A))\end{array}$$

with image in $M^*(A)$ and thus also in $P_D(M^*(A))$:

$$\begin{array}{rcl}\langle st_R\ L,x\rangle & \in & A\\ \langle st_R\ L,x\rangle & \in & M^*(A)\end{array}$$

- $\boxed{M = CLR_{blog} = \{\textbf{ssl}, \textbf{swll}, \textbf{eswlf}\}, \text{ and } D = \{\textbf{edl}_W\}}$

$\boxed{m = \textbf{ssl}}$ The only overlap scenario is of the form

$$\begin{array}{rcl}\langle st_h\ L,x\rangle\langle ld_W\ L',x'\rangle\langle ld_W\ L',x'\rangle & \in & A\\ \langle st_h\ L,x\rangle\langle ld_W\ L',x'\rangle & \in & P_D(A)\\ \langle ld_W\ L',x'\rangle\langle st_h\ L,x\rangle & \in & \textbf{ssl}(P_D(A))\end{array}$$

with image in $P_D(M^*(A))$:

$$\begin{array}{rcl}\langle st_h\ L,x\rangle\langle ld_W\ L',x'\rangle\langle ld_W\ L',x'\rangle & \in & A\\ \langle ld_W\ L',x'\rangle\langle st_h\ L,x\rangle\langle ld_W\ L',x'\rangle & \in & \textbf{ssl}(A)\\ \langle ld_W\ L',x'\rangle\langle ld_W\ L',x'\rangle\langle st_h\ L,x\rangle & \in & \textbf{ssl}(\textbf{ssl}(A))\\ \langle ld_W\ L',x'\rangle\langle st_h\ L,x\rangle & \in & P_D(\textbf{ssl}(\textbf{ssl}(A)))\end{array}$$

$\boxed{m = \textbf{eswlf}}$ The only overlap scenario is of the form

$$\begin{aligned} \langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle\langle ld_W\ L, x\rangle &\in A \\ \langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle &\in P_D(A) \\ \langle st_h\ L, x\rangle &\in \textbf{eswlf}(P_D(A)) \end{aligned}$$

with image in $M^*(A)$ (and thus in $P_D(M^*(A))$):

$$\begin{aligned} \langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle\langle ld_W\ L, x\rangle &\in A \\ \langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle &\in \textbf{eswlf}(A) \\ \langle st_h\ L, x\rangle &\in \textbf{eswlf}(\textbf{eswlf}(A)) \end{aligned}$$

$\boxed{m = \textbf{swll}}$ There are three overlap scenarios (overlap left, overlap right, and overlap both left and right). We only do the third one; the first two are similar.

$$\begin{aligned} &\langle ld_W\ L, x\rangle\langle ld_W\ L, x\rangle \\ &\quad \langle ld_W\ L', x'\rangle\langle ld_W\ L', x'\rangle \in A \\ &\quad \langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle \in P_D(A) \\ &\quad \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle \in \textbf{swll}(P_D(A)) \end{aligned}$$

with image in $P_D(M^*(A))$:

$$\begin{aligned} &\langle ld_W\ L, x\rangle\langle ld_W\ L, x\rangle \\ &\quad \langle ld_W\ L', x'\rangle\langle ld_W\ L', x'\rangle \in A \\ &\langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle \\ &\quad \langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle \in \textbf{swll}(A) \\ &\langle ld_W\ L', x'\rangle\langle ld_W\ L', x'\rangle \\ &\quad \langle ld_W\ L, x\rangle\langle ld_W\ L, x\rangle \in M^*(A) \\ &\quad \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle \in P_D(M^*(A)) \end{aligned}$$

- $\boxed{M = CLR_{blog} = \{\textbf{ssl}, \textbf{swll}, \textbf{eswlf}\}, \text{ and } D = \{\textbf{eds}_W\}}$

$\boxed{m = \textbf{ssl}}$ The only overlap scenario is of the form

$$\begin{aligned} \langle st_W\ L, x\rangle\langle st_W\ L, x\rangle\langle ld_W\ L', x'\rangle &\in A \\ \langle st_W\ L, x\rangle\langle ld_W\ L', x'\rangle &\in P_D(A) \\ \langle ld_W\ L', x'\rangle\langle st_W\ L, x\rangle &\in \textbf{ssl}(P_D(A)) \end{aligned}$$

with image in $P_D(M^*(A))$:

$$\begin{aligned} \langle st_W\ L, x\rangle\langle st_W\ L, x\rangle\langle ld_W\ L', x'\rangle &\in A \\ \langle st_W\ L, x\rangle\langle ld_W\ L', x'\rangle\langle st_W\ L, x\rangle &\in \textbf{ssl}(A) \\ \langle ld_W\ L', x'\rangle\langle st_W\ L, x\rangle\langle st_W\ L, x\rangle &\in \textbf{ssl}(\textbf{ssl}(A)) \\ \langle ld_W\ L', x'\rangle\langle st_W\ L, x\rangle &\in P_D(\textbf{ssl}(\textbf{ssl}(A))) \end{aligned}$$

$\boxed{m = \textbf{swll}}$ There is no overlap scenario.

$\boxed{m = \textbf{eswlf}}$ The only overlap scenario is of the form

$$\begin{aligned} \langle st_W\ L, x\rangle\langle st_W\ L, x\rangle\langle ld_W\ L, x\rangle &\in A \\ \langle st_W\ L, x\rangle\langle ld_W\ L, x\rangle &\in P_D(A) \\ \langle st_W\ L, x\rangle &\in \textbf{eswlf}(P_D(A)) \end{aligned}$$

with image in $P_D(M^*(A))$:

$$\begin{aligned} \langle st_W\ L, x\rangle\langle st_W\ L, x\rangle\langle ld_W\ L, x\rangle &\in A \\ \langle st_W\ L, x\rangle\langle st_W\ L, x\rangle &\in \textbf{eswlf}(A) \\ \langle st_W\ L, x\rangle &\in P_D(\textbf{eswlf}(A)) \end{aligned}$$

- $\boxed{M = CLR_{blog} = \{\textbf{ssl}, \textbf{swll}, \textbf{eswlf}\}, \text{ and } D = \{\textbf{eil}_W\}}$

$\boxed{m = \textbf{ssl}}$ The only overlap scenario is of the form

$$\begin{aligned} &\langle st_h\ L, x\rangle\{\langle ld_W\ L_1, *\rangle\}\cdots \\ &\quad \{\langle ld_W\ L_n, *\rangle\}\langle ld_W\ L', x'\rangle \subseteq A \\ &\quad \langle st_h\ L, x\rangle\langle ld_W\ L', x'\rangle \in P_D(A) \\ &\quad \langle ld_W\ L', x'\rangle\langle st_h\ L, x\rangle \in \textbf{ssl}(P_D(A)) \end{aligned}$$

Now consider the following $w \in A$:

$$w = \langle st_h\ L, x\rangle\langle ld_W\ L_1, x_1\rangle\cdots\langle ld_W\ L_n, x_n\rangle\langle ld_W\ L', x'\rangle$$

where $x_i = x$ whenever $L = L_i$, or arbitrary otherwise. Then, by repeatedly applying **ssl** or **eswlf**, we can move the store all the way to the right, which proves

$$\langle ld_W\ L_{i_1}, x_{i_1}\rangle\cdots\langle ld_W\ L_{i_n}, x_{i_n}\rangle\langle ld_W\ L', x'\rangle\langle st_h\ L, x\rangle \in M^*(A)$$

where $i_1, \ldots, i_n$ is the subsequence of $1, \ldots, k$ consisting of all indices i for which $L_i \neq L$. Finally, because the x_{i_j} were arbitrary, this implies

$$\{\langle ld_W\ L_{i_1}, *\rangle\cdots\langle ld_W\ L_{i_n}, *\rangle\}\langle ld_W\ L', x'\rangle\langle st_h\ L, x\rangle \subseteq M^*(A)$$

and we can thus eliminate the loads, giving

$$\langle ld_W\ L', x'\rangle\langle st_h\ L, x\rangle \subseteq P_D(M^*(A)).$$

$\boxed{m = \textbf{swll}}$ The only overlap scenario is of the form

$$\begin{aligned} &\langle ld_W\ L, x\rangle\{\langle ld_W\ L_1, *\rangle\}\cdots \\ &\quad \{\langle ld_W\ L_n, *\rangle\}\langle ld_W\ L', x'\rangle \subseteq A \\ &\quad \langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle \in P_D(A) \\ &\quad \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle \in \textbf{swll}(P_D(A)) \end{aligned}$$

Now consider the following $w \in A$:

$$w = \langle ld_W\ L, x\rangle\langle ld_W\ L_1, x_1\rangle\cdots\langle ld_W\ L_n, x_n\rangle\langle ld_W\ L', x'\rangle$$

where the x_i are arbitrary. Then, by repeatedly applying **swll**, we can move the loads in the middle all the way to the left. This means

$$\langle ld_W\ L_1, x_1\rangle\cdots\langle ld_W\ L_n, x_n\rangle\langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle \in M^*(A)$$

Because the x_i were arbitrary, this implies

$$\{\langle ld_W\ L_1, *\rangle\} \cdots \{\langle ld_W\ L_n, *\rangle\}\langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle \subseteq M^*(A)$$

and we can thus eliminate the loads, giving

$$\langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle \subseteq P_D(M^*(A)).$$

$\boxed{m = \textbf{eswlf}}$ The only overlap scenario is of the form

$$\begin{array}{rcl} \langle st_h\ L, x\rangle\{\langle ld_W\ L_1, *\rangle\} \cdots \\ \{\langle ld_W\ L_n, *\rangle\}\langle ld_W\ L, x\rangle & \subseteq & A \\ \langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle & \in & P_D(A) \\ \langle st_h\ L, x\rangle & \in & \textbf{eswlf}(P_D(A)) \end{array}$$

Now consider the following $w \in A$:

$$w = \langle st_h\ L, x\rangle\langle ld_W\ L_1, x_1\rangle \cdots \langle ld_W\ L_n, x_n\rangle\langle ld_W\ L, x\rangle$$

where $x_i = x$ whenever $L = L_i$, or arbitrary otherwise. Then, by repeatedly applying **ssl** or **eswlf**, we can move the store all the way to the right and forward the load on the right, which proves

$$\langle ld_W\ L_{i_1}, x_{i_1}\rangle \cdots \langle ld_W\ L_{i_n}, x_{i_n}\rangle\langle st_h\ L, x\rangle \in M^*(A)$$

where $i_1, \ldots, i_n$ is the subsequence of $1, \ldots, k$ consisting of all indices i for which $L_i \neq L$. Finally, because the x_{i_j} were arbitrary, this implies

$$\{\langle ld_W\ L_{i_1}, *\rangle \cdots \langle ld_W\ L_{i_n}, *\rangle\}\langle st_h\ L, x\rangle \subseteq M^*(A)$$

and we can thus eliminate the loads, giving

$$\langle st_h\ L, x\rangle \subseteq P_D(M^*(A))$$

- $\boxed{M = CLR_{blog} = \{\textbf{ssl}, \textbf{swll}, \textbf{eswlf}\}, \text{ and } D = \{\textbf{iil}_W\}}$

$\boxed{m = \textbf{ssl}}$ The only overlap scenario is of the form

$$\begin{array}{rcl} \langle st_h\ L, x\rangle & \in & A \\ \langle st_h\ L, x\rangle\langle ld_W\ L', x'\rangle & \in & P_D(A) \\ \langle ld_W\ L', x'\rangle\langle st_h\ L, x\rangle & \in & \textbf{ssl}(P_D(A)) \end{array}$$

with image in $P_D(M^*(A))$:

$$\begin{array}{rcl} \langle st_h\ L, x\rangle & \in & A \\ \langle st_h\ L, x\rangle & \in & M^*(A) \\ \langle ld_W\ L', x'\rangle\langle st_h\ L, x\rangle & \in & P_D(M^*(A)) \end{array}$$

$\boxed{m = \textbf{swll}}$ There are three overlap scenarios (overlap right, overlap left, and overlap both).

The first overlap scenario is of the form

$$\begin{array}{rcl} \langle ld_W\ L, x\rangle & \in & A \\ \langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle & \in & P_D(A) \\ \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle & \in & \textbf{swll}(P_D(A)) \end{array}$$

with image in $P_D(A)$ and thus in $P_D(M^*(A))$:

$$\begin{array}{rcl} \langle ld_W\ L, x\rangle & \in & A \\ \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle & \in & P_D(A) \end{array}$$

The second overlap scenario is of the form

$$\begin{array}{rcl} \langle ld_W\ L', x'\rangle & \in & A \\ \langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle & \in & P_D(A) \\ \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle & \in & \textbf{swll}(P_D(A)) \end{array}$$

with image in $P_D(A)$ and thus in $P_D(M^*(A))$:

$$\begin{array}{rcl} \langle ld_W\ L', x'\rangle & \in & A \\ \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle & \in & P_D(A) \end{array}$$

The third overlap scenario is of the form

$$\begin{array}{rcl} \epsilon & \in & A \\ \langle ld_W\ L, x\rangle\langle ld_W\ L', x'\rangle & \in & P_D(A) \\ \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle & \in & \textbf{swll}(P_D(A)) \end{array}$$

with image in $P_D(A)$ and thus in $P_D(M^*(A))$:

$$\begin{array}{rcl} \epsilon & \in & A \\ \langle ld_W\ L', x'\rangle\langle ld_W\ L, x\rangle & \in & P_D(A) \end{array}$$

$\boxed{m = \textbf{eswlf}}$ The only overlap scenario is of the form

$$\begin{array}{rcl} \langle st_h\ L, x\rangle & \in & A \\ \langle st_h\ L, x\rangle\langle ld_W\ L, x\rangle & \in & P_D(A) \\ \langle st_h\ L, x\rangle & \in & \textbf{eswlf}(P_D(A)) \end{array}$$

with image in $M^*(A)$ and thus also in $P_D(M^*(A))$:

$$\begin{array}{rcl} \langle st_h\ L, x\rangle & \in & A \\ \langle st_h\ L, x\rangle & \in & M^*(A) \end{array}$$

- $\boxed{M = CLR^{SC}_{blog} = \{\textbf{swswl}, \textbf{swlwl}, \textbf{ewswlf}\}, \text{ and } D = \{\textbf{edl}_W\}}$

same as case $M = CLR_{blog}$ (after replacing rule names and replacing $\langle st_h\ L, x\rangle$ with $\langle st_W\ L, x\rangle$).

- $\boxed{M = CLR^{SC}_{blog} = \{\textbf{swswl}, \textbf{swlwl}, \textbf{ewswlf}\}, \text{ and } D = \{\textbf{eds}_W\}}$

same as case $M = CLR_{blog}$ (after replacing rule names).

- $\boxed{M = CLR^{SC}_{blog} = \{\textbf{swswl}, \textbf{swlwl}, \textbf{ewswlf}\}, \text{ and } D = \{\textbf{eil}_W\}}$

same as case $M = CLR_{blog}$ (after replacing rule names and replacing $\langle st_h\ L, x\rangle$ with $\langle st_W\ L, x\rangle$).

- $\boxed{M = CLR^{SC}_{blog} = \{\textbf{swswl}, \textbf{swlwl}, \textbf{ewswlf}\}, \text{ and } D = \{\textbf{iil}_W\}}$

same as case $M = CLR_{blog}$ (after replacing rule names and replacing $\langle st_h\ L, x\rangle$ with $\langle st_W\ L, x\rangle$).