Information Flow above Optimising Compilers With Weak Memory

Jay Richards, Vineet Rajani, Mark Batty — University of Kent

1 Introduction

Programs routinely process secret information. Information Flow Control (IFC) prevents leaks of such secrets, and entails tracking data and control dependencies. The weak memory semantics provided by concurrent languages like C and C++ allows apparent syntactic dependencies to be ignored in order to permit compiler optimisations [2]. Previous work defines a calculation of dependencies that must remain after optimisation [12]. It is only these so-called *semantic dependencies* that security properties may rely on. Additionally, a program secure under one weak memory semantics may be insecure in another [11]. Prior work develops IFC for processors [18, 15] without a dependency-removing optimiser, and for a concurrent language [16] that excludes weak memory behaviour.

In this work, we present an approach to IFC for compiler optimised code under a varied choice of weak memory semantics, e.g. sequential consistency, release acquire, or C++. The key idea is to use semantic dependencies to abstract away the details of the compiler's transformations and to build a general framework for dependency tracking that is sound with this reduced set of dependencies. This is ongoing work and we are currently in the process of formalising our approach.

We describe the key ideas using the example below. r1, r2, and r3 are thread local variables. L1 and L2 are low security memory locations. H is a high security memory location. Line 1 is the parent thread, lines 2-9 are the first thread and lines 10-12 are the second.

```
1  L1 := 0; L2 := 0; H := secret;

2  r1 := L1;                      10  r3 := L2;
3  if (r1 = 1) {                  11  if (r3 = 1)
4    L2 := 2;                     12    L1 := 1;
5    L2 := 1;
6    r2 := H;
7    L1 := r2;
8  } else
9    L2 := 1;
```

On a weak memory system that forbids optimisations by respecting syntactic dependencies, outcomes where the value at H is leaked are forbidden. In the following, we show that after optimisation this guarantee can be lost.

2 Approach

Our approach uses two pieces of machinery.

A symbolic denotational semantics for weak memory concurrency informs us of valid orderings of atomic accesses with respect to both compiler and hardware optimisations. Its output is a set of *justifications* (appendix A.3), where each justification captures the dependencies of a write. These are computed inductively over an input program. Formally a justification has the following shape:

$$(P, D) \vdash^{\psi} (l : W\ x\ \epsilon)$$

The write being justified is $(l : W\ x\ \epsilon)$, where l is a unique label, x is the memory location being written, and ϵ is the value written. P is a predicate capturing control dependencies, D is a set of read data dependencies, and ψ is a restriction on how the program can execute.

The first step of the semantics generates *initial* justifications from the syntax of the program, e.g. the justification of write 7 records the condition of line 3 in P, and the data dependence on the read of line 6 in D:

$$(\top, \emptyset) \vdash^{\top} (1a : W\ L1\ 0) \quad (\top, \emptyset) \vdash^{\top} (1b : W\ L2\ 0)$$
$$(\top, \emptyset) \vdash^{\top} (1c : W\ H\ secret)$$
$$(r1 = 1, \emptyset) \vdash^{\top} (4 : W\ L2\ 2) \qquad (r1 = 1, \emptyset) \vdash^{\top} (5 : W\ L2\ 1)$$
$$(r1 = 1, \{6\}) \vdash^{\top} (7 : W\ L1\ r2)$$
$$(r1 \neq 1, \emptyset) \vdash^{\top} (9 : W\ L2\ 1)$$
$$(\top, \emptyset) \vdash^{\top} (12 : W\ L1\ 1)$$

The set of justifications is elaborated, e.g. by removing shadowed writes or by lifting writes from branches where the writes happen regardless. For writes at lines 5 and 9, we elide the write at line 4 as it is shadowed by the write on 5, adding (a) and (b) with ψ recording the elision of 4 from *reads from*, a relation linking writes to reads that take their value. Now each branch has an equivalent write of 1 to $L2$, so we lift the writes' dependency on the branch, adding justifications (c) and (d).

$$\text{(a)}\ (r1 = 1, \emptyset) \vdash^{4 \notin \pi_1(rf)} (5 : W\ L2\ 1)$$
$$\text{(b)}\ (r1 \neq 1, \emptyset) \vdash^{4 \notin \pi_1(rf)} (9 : W\ L2\ 1)$$
$$\text{(c)}\ (\top, \emptyset) \vdash^{4 \notin \pi_1(rf)} (5 : W\ L2\ 1) \quad \text{(d)}\ (\top, \emptyset) \vdash^{4 \notin \pi_1(rf)} (9 : W\ L2\ 1)$$

A non-deterministic operational semantics uses the order imposed by justifications to walk the program, ensuring that low writes are independent of high reads. It maintains a state comprising a set of executed events, E, a set of timestamped committed writes to low locations, S, the branch conditions chosen, R, and a predicate restricting the shape of the execution, J (derived from ψ in each justification used in the walk). Subscripts project from a state tuple: X_E, X_S, X_R, X_J. Following justifications contrasts with classical taint analysis where security labels track dependencies syntactically [4, 5, 13].

Def. 2.1. *(State equivalence). Two end states, X and X', of the operational semantics are* equivalent*, written $X \approx X'$, if their low branch conditions are consistent, $X_R \wedge X'_R$, if they have equivalent observable behaviour, $X_S \sim X'_S$, and if the assumptions made about low symbols in resolving justifications are consistent, $X_J \wedge X'_J$.*

We must check all possible high inputs to ensure that no permutation of values leaks secrets. To this end we provide an expression, ϵ, that restricts the values of high symbols i.e. $h1 = 0 \wedge h2 = 0$, $h1 = 0 \wedge h2 = 1$.

Connecting the denotational and operational semantics is a futures function (def. B.1.1), $F^{\mathcal{P}}$, which is instantiated from a program, $\mathcal{P}$. It returns the set of immediately executable instructions as well as the constraints that executing said instruction impose on the shape of the execution.

Def. 2.2. *(Security judgement). For a program, $\mathcal{P}$, and a resulting futures function, $F^{\mathcal{P}}$. For every final state of the operational semantics under restriction ϵ, there is an equivalent end state for restriction ϵ':*

$$\forall \epsilon, \epsilon'.\ (\emptyset, \emptyset, \top, \epsilon) \xrightarrow{fin} X \implies \exists X'.\ (\emptyset, \emptyset, \top, \epsilon') \xrightarrow{fin} X' \wedge X \approx X'$$

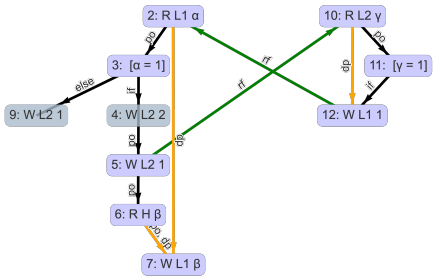
Returning to the example. We must first execute the initialising writes 1a, 1b, and 1c. After this, there is a choice of using justification (c) or (d) to perform the writes at 5 or 9. The read at 2 is unconstrained so that can be chosen also. The futures after executing line 1 are:

$$(5:\ W\ L2\ 1), 4 \notin \pi_1(rf)) \quad ((9:\ W\ L2\ 1), 4 \notin \pi_1(rf))$$
$$((2:\ R\ L1\ r1), \top)$$

If we execute the write at 5, we can now construct an execution where we read 1 into r1 thus allowing the information leak. States X and X' are reachable in the operational semantics, violating Def. 2.2:

$$(1, (L1, 41), r1 = 1) \in X_S \qquad (1, (L1, 42), r1 = 1) \in X'_S \qquad X_S \not\approx X'_S$$

Below is a weak memory execution of this program: rf connects writes to reads that read that value, and dp – derived from justification – connects reads to writes which are dependent on them. Event 4 is elided and event 9 is on the untaken branch.



The above execution is valid under the C++11, POWER, and ARM memory models. However it is forbidden under sequential consistency, as reordering the write at 4 prior to the read at 2 is disallowed. We introduce an *extended futures* function (def. B.1.2), $F^{\mathcal{P}}_M$, that filters the allowed futures based upon whether they would constitute a valid ordering under the memory model, M; this function replaces all prior uses within the operational semantics and we also restrict the final state according to M (appendix B.3).

The approach presented here identified a failure of IFC in our example caused by the interaction of weak memory concurrency and compiler optimisations. We are working towards verifying that an algorithm is secure within this environment: if an algorithm does not violate our security judgement then we assert that there is non-interference.

Compositionality is an important part of any concurrent semantics. Justification is calculated per-thread in our denotational semantics, so our security judgement, driven by justification, need only be considered on a per-thread basis until executions are projected.

3 Related Work

Relaxed memory. The C/C++ memory model permits dubious *thin-air* behaviours [3], and reasoning under a memory model that permits these behaviours is impossible [2]. Work has been done to produce memory models that do not exhibit these behaviours [12, 9, 10, 8], but they fail by restricting the allowed executions too much, invalidating compiler optimisations.

MRD [12] is a thin-air free memory model that allows a wide variety of common compiler optimisations, this is the base from which the dependency component of our semantics is built upon. *All* current thin-air free memory models fail to accommodate some compiler optimisations, and none has been taken up by the C++ specification, so there is no settled decision on which optimisations should be allowed. Our approach separates the concerns of the memory orderings from the security guarantees so that these components can be separately proved and updated, having an exportable justification allows for this.

Promising [9] is a thin-air free memory model that uses an operational semantics to walk traces of the program reasoning about valid orderings along the way with *promises*. However it suffers in that it is undecidable [1], and that it violates the C/C++ coherence rules [6].

Information flow security. An optimising compiler is unaware of the security requirements of the underlying source code, and therefore it does not attempt to preserve them, this is known as the correctness-security gap [7]; this paper worked through several examples of cases where security guarantees are invalidated by optimisations and also posed many questions around what should be done in regards to addressing this gap. It should be mentioned that the breakdown of guarantees is further exacerbated under a concurrent environment – as the *as-if* rule is a thread local guarantee that is not necessarily maintained in such an environment. Of the questions posed we hope to tackle *Generalised compiler correctness proofs*, *Testing tools*, and *Correctness-Security Violation Detectors* with respect to a weak memory environment.

A separation logic has been created for the Promising semantics [17], it has similar guarantees to that of RSL [19] with novel features for reasoning about thin-air freedom and coherence. But the promising model does not accommodate all needed optimisations.

Smith et al. [14, 15] created a program logic for value-dependent information flow control under weak orderings of any multi-copy-atomic memory model; they instantiate it with ARM and POWER models. It takes into account hardware optimisations and reorderings allowed by inter-instruction dependencies. They provide a tool to automate the application of this logic to a program.

References

[1] Abdulla, P. A., Atig, M. F., Godbole, A., Krishna, S., and Vafeiadis, V. The decidability of verification under PS 2.0. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings* (2021), N. Yoshida, Ed., vol. 12648 of *Lecture Notes in Computer Science*, Springer, pp. 1–29.

[2] Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., and Sewell, P. The problem of programming language concurrency semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings* (2015), J. Vitek, Ed., vol. 9032 of *Lecture Notes in Computer Science*, Springer, pp. 283–307.

[3] Batty, M., Owens, S., Sarkar, S., Sewell, P., and Weber, T. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 55–66.

[4] Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014* (2014), A. Russo and O. Tripp, Eds., ACM, p. 15.

[5] Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings* (2014), M. Abadi and S. Kremer, Eds., vol. 8414 of *Lecture Notes in Computer Science*, Springer, pp. 159–178.

[6] Chakraborty, S., and Vafeiadis, V. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang. 3*, POPL (2019), 70:1–70:28.

[7] D'Silva, V., Payer, M., and Song, D. X. The correctness-security gap in compiler optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015* (2015), IEEE Computer Society, pp. 73–87.

[8] Jeffrey, A., and Riely, J. On thin air reads: Towards an event structures model of relaxed memory. *Log. Methods Comput. Sci. 15*, 1 (2019).

[9] Kang, J., Hur, C., Lahav, O., Vafeiadis, V., and Dreyer, D. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 175–189.

[10] Lee, S., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C., Lahav, O., and Vafeiadis, V. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020* (2020), A. F. Donaldson and E. Torlak, Eds., ACM, pp. 362–376.

[11] Mantel, H., Perner, M., and Sauer, J. Noninterference under weak memory models. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014* (2014), IEEE Computer Society, pp. 80–94.

[12] Pavoitti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., and Batty, M. Modular relaxed dependencies in weak memory concurrency. In *Programming Languages and Systems* (Cham, 2020), P. Müller, Ed., Springer International Publishing, pp. 599–625.

[13] Rajani, V., Bichhawat, A., Garg, D., and Hammer, C. Information flow control for event handling and the DOM in web browsers. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015* (2015), C. Fournet, M. W. Hicks, and L. Viganò, Eds., IEEE Computer Society, pp. 366–379.

[14] Smith, G., Coughlin, N., and Murray, T. Value-dependent information-flow security on weak memory models. In *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings* (2019), M. H. ter Beek, A. McIver, and J. N. Oliveira, Eds., vol. 11800 of *Lecture Notes in Computer Science*, Springer, pp. 539–555.

[15] Smith, G., Coughlin, N., and Murray, T. Information-flow control on ARM and POWER multicore processors. *Formal Methods Syst. Des. 58*, 1-2 (2021), 251–293.

[16] Smith, G., and Volpano, D. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), POPL '98, Association for Computing Machinery, p. 355–364.

[17] Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., and Vafeiadis, V. A separation logic for a promising semantics. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings* (2018), A. Ahmed, Ed., vol. 10801 of *Lecture Notes in Computer Science*, Springer, pp. 357–384.

[18] Vaughan, J. A., and Millstein, T. D. Secure information flow for concurrent programs under total store order. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012* (2012), S. Chong, Ed., IEEE Computer Society, pp. 19–29.

[19] Yan, P., and Murray, T. Secrsl: security separation logic for C11 release-acquire concurrency. *Proc. ACM Program. Lang. 5*, OOPSLA (2021), 1–26.

A Dependency calculation

A.1 Language

A type, $t \in \mathcal{T}$, is defined as a set of values, $\mathbb{V}(t)$, and a set of operators.

$$\mathcal{V} = \bigcup \forall t \in \mathcal{T}.\ \mathbb{V}(t)$$

Expressions, $\mathcal{E}$, are either: a symbolic value, $s \in \mathcal{S}$, a concrete value, $v \in \mathcal{V}$, or expressions composed with an operator.

$$\mathcal{E} = s \mid \mathcal{E}\ binaryoperator\ \mathcal{E} \mid unaryoperator\ \mathcal{E} \mid v$$

For an instance of an expression, ϵ, $symbols(\epsilon)$ is the minimal subset of $\mathcal{S}$ required to construct ϵ, $\epsilon|S$ is the minimal evaluatable expression that only contains symbols in S.

An event is one of:

- A write, $W\ x\ \epsilon$, where x is the location being written to, and ϵ is the expression written.
- A read, $R\ x\ s$, where x is the location being written to, and s is the symbol introduced by performing this read.
- A memory fence, F_o, where o is the ordering that is imposed by the fence e.g. SC, REL, ACQ.
- A branch, $[\epsilon]$, where ϵ is the expression which we branch upon.

$$loc(e) = x.\ e \in \{W\ x\ \epsilon,\ R\ x\ s\} \qquad val(e) = \epsilon.\ e \in \{W\ x\ \epsilon,\ R\ x\ \epsilon,\ [\epsilon]\}$$
$$O(s) = e.\ e \in \mathcal{R} \wedge val(e) = s \qquad O(\epsilon) = \bigcup_{s \in symbols(\epsilon)} O(s)$$

A program is made up of a set of labelled events, $\mathbb{E}$. A labelled event, $(l : e)$, extends an event with a unique label, $l \in \mathcal{L}$.

$$label(e) = l.\ (l :\ e)$$

Several maximal subsets are defined on this set: $\mathcal{W}$, the set of all writes, $\mathcal{R}$, the set of all reads, $\mathcal{F}$, the set of all fences, $\mathcal{B}$, the set of all branches. Each of these sets can be restricted to high, $\mathbb{H}$, and low, $\mathbb{L}$, locations:

$$(e \in \mathbb{E}\ \wedge\ loc(e) \in \mathbb{L}) \implies e \in \mathbb{E}_{\mathbb{L}} \qquad (e \in \mathbb{E}\ \wedge\ loc(e) \in \mathbb{H}) \implies e \in \mathbb{E}_{\mathbb{H}}$$

$\mathbb{P}(l)$ is the syntactic control dependencies for a labelled event $(l : e)$.

A.2 Event structure

A program written in our language is consumed in a continuation passing style to create an *event structure*, from this *justifications* are calculated.

A.3 Justification

For every write present in an event structure we create a justification. *Initial justifications* are generated using purely syntactic constraints. From here we apply a sequence of operations that determine which of these are not true dependencies and we project out justifications which no longer have these constraints applied, the details of these operations are part of concurrent work.

Def. A.3.1. *(Justification).*

$$(P, D) \vdash^{(\psi,\ \leq,\ \mathbb{E})} w$$
$$D \in \mathbb{P}(\mathcal{L})$$
$$w \in \mathbb{E}_{\mathcal{W}}$$

A justification contains:

- Control dependencies, P, a predicate over the symbol environment that must be met for the write to happen.
- Data dependencies, D, a set of the symbol values that are present in the written value, along with any that are introduced as part of the dependency calculation.

- Execution restrictions, ψ, a predicate restricting the executions that this justification is valid in – an important distinction between this and P is that ψ does not introduce any ordering constraints on when the write can execute.
- Preserved program order, $\leq$, a subset of the syntactic program order that is guaranteed to be preserved. This contains edges between accesses to the same locations as well as those introduced by pointer aliasing and synchronisation instructions.
- Program, $\mathbb{E}$, optimisations that make write values concrete, promote events to their synchronised equivalent or otherwise transform individual events have a different understanding of the program, $\mathbb{E}$ holds this information. The need for this becomes apparent when you consider optimisations that are performed based on guarantees not present in the source such as: undefined behaviour or alignment.
- A write, $(l :\ W\ x\ \epsilon)$, which is being justified.

B Security

B.1 Futures

The futures function, $F^{\mathcal{P}}$, takes the current state of the execution and computes the events that can be executed next. It operates on the justification set, $\mathbb{F}^{\mathcal{P}}$, obtained from consuming program $\mathcal{P}$.

Def. B.1.1. *(Futures).*

$$F^{\mathcal{P}}(E,R,J) = \bigcup_{(P,D)\vdash^{(\psi,\leq,\mathbb{E})} w\ \in\ \mathbb{F}^{\mathcal{P}}} \{\ (e,\psi)\ |\ consistent(e) \wedge \begin{cases} (R\implies P) \wedge D \subseteq E & e = w \\ O(val(e)) \subseteq E & e \in \mathbb{E}_{\mathcal{B}} \\ \top & e \in (\mathbb{E}_{\mathcal{R}} \cup \mathbb{E}_{\mathcal{F}}) \end{cases} \}$$

$$consistent(e)\ =\ (J \wedge \psi)_{[rf]} \wedge label(e) \notin E \wedge ppoConsistent(e) \qquad rf = (E \cup \{l\}) \times (E \cup \{l\})$$
$$ppoConsistent(e)\ =\ \nexists\ l'.\ l' \leq label(e) \wedge l' \notin E$$

Def. B.1.2. *(Extended Futures). The extended futures function, $F^{\mathcal{P}}_M$, filters the futures returned by the futures function depending upon whether a memory model, M, allows this within its ordering constraints.*

$$F^{\mathcal{P}}_M(E,R,J) = \{f\ |\ f \in F^{\mathcal{P}}(E,R,J) \wedge M(E \cup \{\pi_1(f)\},R,J)\}$$

B.2 Operational Semantics

The initial state for our operational semantics is:

$$(E,S,R,J) = (\emptyset,\emptyset,\top,\top)$$

With E containing the executed event labels, S containing our committed stores to low locations, R containing the choices made when branching, and J containing the combined restrictions over the execution taken from justifications. We can freely strengthen our execution restriction. Our operational semantics transition is:

$$(\mathcal{L} \times \mathbb{S} \times \mathcal{E} \times \mathcal{E}) \xrightarrow{fin} (\mathbb{S} \times \mathcal{E} \times \mathcal{E}) \qquad \mathbb{S} = \mathbb{N} \times (loc, \mathcal{E}) \times \mathcal{E}$$

Def. B.2.1. *(Committed store isomorphism). Two sets of committed writes, A and B, are isomorphic iff for every ordering of equivalent writes in one, there exists an equivalent ordering in the other. The function $(\xrightarrow{\Lambda})$ maps symbols introduced in one branch to equivalent symbols in another branch, it is required in order to compare symbols introduced in one branch with those introduced in another, the definition is omitted.*

$$A \sim B = B \sim A \wedge \forall\ (t,(x,\epsilon_a),R_a) \in A.\ \exists\ (t,(x,\epsilon_b),R_b) \in B.\ (R_a \wedge R_b\implies \epsilon_a = (\xrightarrow{\Lambda})\epsilon_b)$$

$$\text{FINISH}\frac{F^{\mathcal{P}}(E,R,J) = \emptyset \wedge J}{(E,S,R,J) \xrightarrow{fin} (E,S,R|_{\mathbb{L}},J|_{\mathbb{L}})}$$

$$\text{FENCE}\frac{((l:\ F_o),\ \psi) \in F^{\mathcal{P}}(E,R,J)}{(E,S,R,J) \to (E \cup \{l\},S,R,J \wedge \psi)} \qquad \text{LOAD}\frac{((l:\ e),\ \psi) \in F^{\mathcal{P}}(E,R,J) \wedge e \in \mathcal{R}}{(E,S,R,J) \to (E \cup \{l\},S,R,J \wedge \psi)}$$

$$\text{HSTORE}\frac{((l:\ W\ x\ \epsilon),\ \psi) \in F^{\mathcal{P}}(E,R,J) \quad x \in \mathbb{H}}{(E,S,R,J) \to (E \cup \{l\},S,R,J \wedge \psi)} \qquad \text{LSTORE}\frac{\begin{array}{c}((l:\ W\ x\ \epsilon),\ \psi) \in F^{\mathcal{P}}(E,R,J) \quad x \in \mathbb{L} \\ \forall t' \in \pi_1(S).\ t > t' \qquad S' = \{(t,(x,\epsilon),R)\}\end{array}}{(E,S,R,J) \to (E \cup \{l\},S \cup S',R,J \wedge \psi)}$$

$$\text{BRANCH-T}\frac{((l:\ [\epsilon]),\ \psi) \in F^{\mathcal{P}}(E,R,J)}{(E,S,R,J) \to (E \cup \{l\},S,R \wedge \epsilon,J \wedge \psi)} \qquad \text{BRANCH-F}\frac{((l:\ [\epsilon]),\ \psi) \in F^{\mathcal{P}}(E,R,J)}{(E,S,R,J) \to (E \cup \{l\},S,R \wedge \neg\epsilon,J \wedge \psi)}$$

B.3 Extended operational semantics

In our extended operational semantics all uses of our futures function, $F^{\mathcal{P}}$, are replaced with our extended futures function, $F^{\mathcal{P}}_M$, and we introduce a new precondition to the FINISH rule.

$$\text{FINISH}_M\frac{F^{\mathcal{P}}_M(E,R,J) = \emptyset \wedge J \wedge M(E,R,J)}{(E,S,R,J) \xrightarrow{fin} (E,S,R|_{\mathbb{L}},J|_{\mathbb{L}})}$$