# A model of speculative evaluation

CRAIG DISSELKOEN, University of California San Diego
RADHA JAGADEESAN, DePaul University
ALAN JEFFREY, Mozilla Research
JAMES RIELY, DePaul University

## 1 INTRODUCTION

## 2 MODEL

### 2.1 Preliminaries

We assume:

- a set of *memory locations* $\mathcal{X}$, ranged over by $x$ and $y$,
- a set of *registers* $\mathcal{R}$, ranged over by $r$ and $s$,
- a set of *values* $\mathcal{V}$, ranged over by $v$ and $w$,
- a set of *expressions* $\mathcal{E}$, ranged over by $M$ and $N$,
- a set of *logical formulae* $\Phi$, ranged over by $\phi$ and $\psi$, and
- a set of *actions* $\mathcal{A}$, ranged over by $a$ and $b$,

such that:

- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions are closed under substitution, written $M[N/r]$,
- formulae include at least true, false, and equalities of the form $(M = N)$ and $(x = N)$,
- formulae are closed under negation, conjunction, disjunction,
- formulae are closed under substitution, written $\phi[N/\ell]$, and
- there are relations $\mathsf{R}$ and $\mathsf{W} \subseteq (\mathcal{A} \times \mathcal{X} \times \mathcal{V})$,

where:

- the set of *lvalues* is $\mathcal{L} = (\mathcal{X} \cup \mathcal{R})$, ranged over by $\ell$ and $k$.

We shall say *a reads v from x* whenever $(a, x, v) \in \mathsf{R}$, and *a writes v to x* whenever $(a, x, v) \in \mathsf{W}$. In examples, the actions are of the form $(\mathsf{R}\, x\, v)$, which reads $v$ from $x$, and $(\mathsf{W}\, x\, v)$, which writes $v$ to $x$.

### 2.2 Pomsets

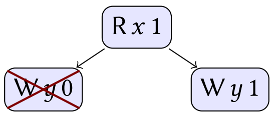
*Definition 2.1.* A *pomset* $(E, \leq, \lambda)$ with alphabet $\Sigma$ is a partial order $(E, \leq)$ together with a function $\lambda : E \to \Sigma$.

Authors' addresses: Craig Disselkoen, University of California San Diego, cdisselk@eng.ucsd.edu; Radha Jagadeesan, DePaul University, rjagadeesan@cs.depaul.edu; Alan Jeffrey, Mozilla Research, ajeffrey@mozilla.com; James Riely, DePaul University, jriely@cs.depaul.edu.

XXXX-XXXX/2018/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Going forward, we fix the alphabet $\Sigma = (\Phi \times \mathcal{A})$. We will write $(\phi \mid a)$ for the pair $(\phi, a)$, $a$ for $(\mathsf{true}, a)$ and $\cancel{a}$ for $(\mathsf{false}, a)$.

We visualize a pomset as a graph where the nodes are drawn from $E$, each node $e$ is labelled with $\lambda(e)$, and an edge $d \to e$ corresponds to an ordering $d \le e$. For example:



is a visualization of the pomset where:

$$0 \le 1 \quad 0 \le 2 \quad \lambda(0) = (\mathsf{true}, \mathsf{R}\,x\,1) \quad \lambda(1) = (\mathsf{false}, \mathsf{W}\,y\,0) \quad \lambda(2) = (\mathsf{true}, \mathsf{W}\,y\,1)$$

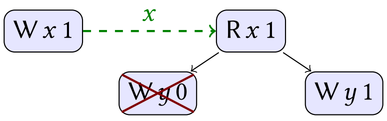
As we shall see, this is a possible execution of the program:

```
r := x; if (r) { y := 1; } else { y := 0; }
```

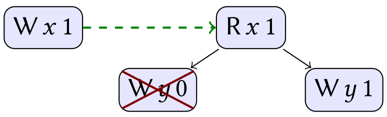
*Definition 2.2.* An *rf-pomset* is a pomset together with a $\mathsf{RF} \subseteq (E \times \mathcal{X} \times E)$ such that for any $(d, x, e) \in \mathsf{RF}$:

- $d < e$,
- $\lambda(d)$ writes $v$ to $x$, and $\lambda(e)$ reads $v$ from $x$, and
- there is no $d < c < e$ such that $\lambda(c)$ writes $w$ to $x$.

We visualize rf-pomsets by drawing a dashed edge between nodes in RF, labelled with the memory location, for example:



In most cases, the memory location is obvious from context, so we elide it:



As we shall see, this is a possible execution of the program:

```
x := 1;  ||  r := x; if (r) { y := 1; } else { y := 0; }
```

*Definition 2.3.* An rf-pomset is $x$-closed if for $e \in E$ with $\lambda(e) = (\phi \mid a)$:

- $\phi$ is independent of $x$, and
- if $a$ reads $v$ from $x$, then there is a $d$ with $(d, x, e) \in \mathsf{RF}$.

## 2.3 Sets of pomsets

Let $\mathcal{P}_1 \sqcup \mathcal{P}_2$ be the set $\mathcal{P}_0$ where $P_0 \in \mathcal{P}_0$ whenever there are $P_1 \in \mathcal{P}_1$ and $P_2 \in \mathcal{P}_2$ such that:

- $E_0 = E_1 \cup E_2$,
- $\mathsf{RF}_0 = \mathsf{RF}_1 \cup \mathsf{RF}_2$,
- if $e \le_1 d$ or $e \le_2 d$ then $e \le_0 d$,
- if $\lambda_0(e) = (\phi_0 \mid a)$ then either:
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $\lambda_2(e) = (\phi_2 \mid a)$ and $\phi_0$ implies $\phi_1 \vee \phi_2$,
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $e \notin E_2$ and $\phi_0$ implies $\phi_1$, or
  - $\lambda_2(e) = (\phi_2 \mid a)$ and $e \notin E_1$ and $\phi_0$ implies $\phi_2$.

Let $\mathcal{P}_1 \parallel \mathcal{P}_2$ be defined the same as $\mathcal{P}_1 \sqcup \mathcal{P}_2$ except that:

- $\mathsf{RF}_0 \supseteq \mathsf{RF}_1 \cup \mathsf{RF}_2$, and for any $d, e \in E_i$, if $(d, e) \in \mathsf{RF}_0$ then $(d, e) \in \mathsf{RF}_i$.

Let $(\phi \mid a) \to \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

- $E' = E \cup \{0\}$,
- $\mathsf{RF}' = \mathsf{RF}$,
- if $d \le e$ then $d \le' e$,
- $\lambda'(0) = (\phi, a)$, where $\psi$ implies $\phi$, and
- if $\lambda(e) = (\psi \mid b)$ then:
  - $\lambda'(e) = (\psi' \mid b)$,
  - $\psi'$ implies $\psi[\vec{v}/\vec{x}]$, where $a$ reads $\vec{v}$ from $\vec{x}$, and
  - $0 \le' e$ or $\psi'$ implies $\psi$.

Let $\mathcal{P}[M/\ell]$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

- $E' = E$,
- $\mathsf{RF}' = \mathsf{RF}$,
- if $d \le e$ then $d \le' e$, and
- if $\lambda(e) = (\psi \mid a)$ then $\lambda'(e) = (\psi[M/\ell] \mid a)$.

Let $(\phi \mid \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever:

- if $\lambda(e) = (\psi \mid a)$ then $\phi$ implies $\psi$.

Let $(\nu x \,.\, \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever $P$ is $x$-closed.

### 2.4 Semantics of programs

Define:

$$
\begin{array}{rcl}
[\![\mathtt{skip}]\!] & = & \{\emptyset\} \\
[\![x \mathbin{:=} M;\, E]\!] & = & \bigcup_{v} (M = v \mid \mathsf{W}\,x\,v) \to [\![E]\!][M/x] \\
[\![r \mathbin{:=} x;\, E]\!] & = & [\![E]\!][x/r] \cup \bigcup_{v} (\mathsf{true} \mid \mathsf{R}\,x\,v) \to [\![E]\!][x/r] \\
[\![\mathtt{if}\, M\, \mathtt{then}\, E\, \mathtt{else}\, D]\!] & = & (M \neq 0 \mid [\![E]\!]) \sqcup (M = 0 \mid [\![D]\!]) \\
[\![E \mathbin{||} D]\!] & = & [\![E]\!] \parallel [\![D]\!] \\
[\![\mathtt{var}\, x;\, E]\!] & = & \nu x \,.\, [\![E]\!]
\end{array}
$$

## 3 EXAMPLES

### 3.1 Sequential memory accesses

In the semantics of memory, there are two very different ways memory can be accessed: sequentially or concurrently. These are modelled differently, since hardware and compilers give very different guarantees about their behaviour. In this section, we discuss the sequential semantics, and leave the concurrent semantics to §3.2.

Consider the program `(x := 0; y := x+1;)`. One execution of this program is where the write to $y$ uses the sequential value of $x$, which is 0:

$$\boxed{\mathsf{W}\,x\,0} \qquad \boxed{\mathsf{W}\,y\,1}$$

To see how this execution is modelled, we first expand out the syntax sugar to get the program `(x := 0; r := x; y := r+1; skip)`. Now $[\![\mathsf{skip}]\!]$ is just $\{\emptyset\}$, and $[\![y := r+1;\ \mathsf{skip}]\!]$ is:

$$\bigcup_v (r+1 = v \mid \mathsf{W}\, y\, v) \rightarrow [\![\mathsf{skip}]\!][v/y]$$

which includes the case where $v$ is 1:

$$(r+1 = 1 \mid \mathsf{W}\, y\, 1) \rightarrow [\![\mathsf{skip}]\!][1/y]$$

which contains the pomset:

$$\boxed{r+1 = 1 \mid \mathsf{W}\, y\, 1}$$

Now $[\![r := x;\ y := r+1;\ \mathsf{skip}]\!]$ is:

$$[\![y := r+1;\ \mathsf{skip}]\!][x/r] \cup \bigcup_v (\mathsf{R}\, x\, v) \rightarrow [\![y := r+1;\ \mathsf{skip}]\!][x/r]$$

This has two cases, the sequential case (which does not introduce a read action) and the concurrent case (which does). For the moment, we are interested in the sequential case, which is:

$$[\![y := r+1;\ \mathsf{skip}]\!][x/r]$$

which contains the pomset:

$$\boxed{x+1 = 1 \mid \mathsf{W}\, y\, 1}$$

Finally $[\![x := 0;\ r := x;\ y := r+1;\ \mathsf{skip}]\!]$ is:

$$\bigcup_v (0 = v \mid \mathsf{W}\, x\, v) \rightarrow [\![r := x;\ y := r+1;\ \mathsf{skip}]\!][v/x]$$

which includes the case where $v$ is 0:

$$(0 = 0 \mid \mathsf{W}\, x\, 0) \rightarrow [\![r := x;\ y := r+1;\ \mathsf{skip}]\!][0/x]$$

which contains the pomset:

$$\boxed{0 = 0 \mid \mathsf{W}\, x\, 0} \quad \boxed{0+1 = 1 \mid \mathsf{W}\, y\, 1}$$

all of whose preconditions are tautologies, so this has the expected behaviour:

$$\boxed{\mathsf{W}\, x\, 0} \quad \boxed{\mathsf{W}\, y\, 1}$$

Note that $\mathsf{W}\, x\, 0$ does not read anything, and so there is no requirement of order between $(\mathsf{W}\, x\, 0)$ and $(\mathsf{W}\, y\, 1)$.

This example demonstrates how preconditions capture the sequential semantics of memory. In an execution containing an event with label $(\phi \mid a)$, one way the precondition $\phi$ can be discharged is by a write $x := M$, which performs a substitution $[M/x]$. This is a variant of the usual Hoare semantics for assignment, where if $E$ has preconditon $\phi$ then $x := M;\ E$ has precondition $\phi[M/x]$.

## 3.2 Concurrent memory accesses

## 3.3 No dependencies on writes

Consider an example with two independent writes (`x := 1; y := 2;`). This has the semantics:

$$\bigcup_v (1 = v \mid \mathsf{W}\,x\,v) \rightarrow \left( \bigcup_w (2 = w \mid \mathsf{W}\,y\,v) \rightarrow (\{\emptyset\})\,[2/y] \right) [1/x]$$

which is the same as:

$$\bigcup_v (1 = v \mid \mathsf{W}\,x\,v) \rightarrow \bigcup_w (2 = w \mid \mathsf{W}\,y\,v) \rightarrow \{\emptyset\}$$

which includes the case where $v = 1$ and $w = 2$:

$$(1 = 1 \mid \mathsf{W}\,x\,1) \rightarrow (2 = 2 \mid \mathsf{W}\,y\,2) \rightarrow \{\emptyset\}$$

One of the executions this contains is:

$$\begin{tikzcd} \boxed{\mathsf{W}\,x\,1} \arrow[r] & \boxed{\mathsf{W}\,y\,2} \end{tikzcd}$$

but it also contains:

$$\begin{tikzcd} \boxed{\mathsf{W}\,x\,1} & \boxed{\mathsf{W}\,y\,2} \end{tikzcd}$$

and:

$$\begin{tikzcd} \boxed{\mathsf{W}\,x\,1} & \boxed{\mathsf{W}\,y\,2} \arrow[l] \end{tikzcd}$$

since there is no requirement that $(\mathsf{W}\,x\,1) \leq (\mathsf{W}\,y\,2)$.

Thus, the semantics of (`x := 1; y := 2;`) is the same as the semantics of (`y := 2; x := 1;`).

## 3.4 Dependencies on reads

Whereas write prefixing introduces no new dependencies, read prefixing can. For example the program (`r := x; y := r+1;`) has semantics:

$$\bigcup_v (\mathsf{true} \mid \mathsf{R}\,x\,v) \rightarrow \left( \bigcup_w (r + 1 = w \mid \mathsf{W}\,y\,v) \rightarrow (\{\emptyset\})\,[r + 1/y] \right) [x/r]$$

which is the same as:

$$\bigcup_v (\mathsf{true} \mid \mathsf{R}\,x\,v) \rightarrow \bigcup_w (x + 1 = w \mid \mathsf{W}\,y\,v) \rightarrow \{\emptyset\}$$

which includes the case where $v = 1$ and $w = 2$:

$$(\mathsf{true} \mid \mathsf{R}\,x\,1) \rightarrow (x + 1 = 2 \mid \mathsf{W}\,y\,2) \rightarrow \{\emptyset\}$$

Now, since true implies $(x + 1 = 2)[1/x]$, this contains:

$$\begin{tikzcd} \boxed{\mathsf{R}\,x\,1} \arrow[r] & \boxed{\mathsf{W}\,y\,2} \end{tikzcd}$$

but since true does not imply $(x + 1 = 2)$, we have the requirement that $(\mathsf{R}\,x\,1) \leq (\mathsf{W}\,y\,2)$.

This is in contrast to the program (`r := x; y := r+2-r;`), which has semantics:

$$\bigcup_v (\mathsf{true} \mid \mathsf{R}\,x\,v) \rightarrow \bigcup_w (x + 2 - x = w \mid \mathsf{W}\,y\,v) \rightarrow \{\emptyset\}$$

Again, we consider the case where $v = 1$ and $w = 2$:

$$(\mathsf{true} \mid \mathsf{R}\,x\,1) \rightarrow (x + 2 - x = 2 \mid \mathsf{W}\,y\,2) \rightarrow \{\emptyset\}$$

Now, since true implies $(x + 2 - x = 2)[1/x]$, this contains:

$$\begin{tikzcd} \boxed{\mathsf{R}\,x\,1} \arrow[r] & \boxed{\mathsf{W}\,y\,2} \end{tikzcd}$$

but also true implies $(x + 2 - x = 2)$ (at least for arithmetic modulo $n$) and so this also contains:

$$\begin{tikzcd} \boxed{\mathsf{R}\,x\,1} & \boxed{\mathsf{W}\,y\,2} \end{tikzcd}$$

and:

$$\begin{tikzcd} \boxed{\mathsf{R}\,x\,1} & \boxed{\mathsf{W}\,y\,2} \arrow[l] \end{tikzcd}$$

Thus, the semantics of (`r := x; y := x+2-x;`) is the same as the semantics of (`y := 2; r := x;`).

## 3.5 Dependencies on conditionals

Conditionals introduce control dependencies, for example consider the program:

```
  r := z; if (r) { x := 1; } else { y := 2; }
```

This has semantics:

$$\bigcup_{v} (\text{true} \mid \mathsf{R}\,z\,v) \rightarrow (z \neq 0 \mid [\![\texttt{x := 1}]\!]) \sqcup (z = 0 \mid [\![\texttt{y := 2}]\!])$$

Now $[\![\texttt{x := 1}]\!]$ contains any pomset of the form:

$$\boxed{\phi \mid \mathsf{W}\,x\,1}$$

in particular it contains:

$$\boxed{r \neq 0 \mid \mathsf{W}\,x\,1}$$

and similarly $[\![\texttt{y := 2}]\!]$ contains any pomset of the form:

$$\boxed{r = 0 \mid \mathsf{W}\,y\,2}$$

and so $[\![\texttt{if (r) \{ x := 1; \} else \{ y := 2; \}}]\!]$ contains:

$$\begin{tikzcd} \boxed{r \neq 0 \mid \mathsf{W}\,x\,1} & \boxed{r = 0 \mid \mathsf{W}\,y\,2} \end{tikzcd}$$

Now, prefixing with $(\mathsf{R}\,z\,v)$ performs substitutions, for example when $v$ is 0 we have:

$$\begin{tikzcd} \boxed{\mathsf{R}\,z\,0} \arrow[r] \arrow[rr, bend left] & \cancel{\boxed{\mathsf{W}\,x\,1}} & \boxed{\mathsf{W}\,y\,2} \end{tikzcd}$$

and when $v$ is 1 we have:

$$\begin{tikzcd} \boxed{\mathsf{R}\,z\,1} \arrow[r] \arrow[rr, bend left] & \boxed{\mathsf{W}\,x\,1} & \cancel{\boxed{\mathsf{W}\,y\,2}} \end{tikzcd}$$

## 3.6 Relaxed memory

The model includes concurrent memory accesses, which can introduce concurrent reads-from. For example, the program:

```
  x := 1; || r := x;
```

has an execution in which the write to $x$ does not justify the read from $x$:

$$\begin{tikzcd} \boxed{\mathrm{W}\,x\,1} & \boxed{\mathrm{R}\,x\,v} \end{tikzcd}$$

but also has an execution in which the write to $x$ does justify the read from $x$:

$$\begin{tikzcd} \boxed{\mathrm{W}\,x\,1} \arrow[r, dashed] & \boxed{\mathrm{R}\,x\,1} \end{tikzcd}$$

Since we are allowing events to be partially ordered, this gives a simple model of relaxed memory, for example an independent read independent write (IRIW) example is:

```
  x := 0; x := x+1; || y := 0; y := y+1; || if (x) { r := y; } || if (y) { s := x; }
```

which includes the execution:

$$\begin{tikzcd} \boxed{\mathrm{W}\,x\,0} \arrow[r] \arrow[rrrrrrr, dashed, bend right] & \boxed{\mathrm{W}\,x\,1} \arrow[rrr, dashed, bend left] & \boxed{\mathrm{W}\,y\,0} \arrow[r] \arrow[rrr, dashed, bend left] & \boxed{\mathrm{W}\,y\,1} \arrow[rrr, dashed, bend right] & \boxed{\mathrm{R}\,x\,1} \arrow[r] & \boxed{\mathrm{R}\,y\,0} & \boxed{\mathrm{R}\,y\,1} \arrow[r] & \boxed{\mathrm{R}\,x\,0} \end{tikzcd}$$

This model does not introduce thin-air reads (TAR), for example the TAR pit is:

```
  x := y; || y := x;
```

but an attempt to produce a value from thin air fails, for the usual reason of producing a cycle in $\leq$:

$$\begin{tikzcd} \boxed{\mathrm{R}\,y\,42} \arrow[d] & \boxed{\mathrm{R}\,x\,42} \arrow[d] \\ \boxed{\mathrm{W}\,x\,42} \arrow[ur, dashed] & \boxed{\mathrm{W}\,y\,42} \arrow[ul, dashed] \end{tikzcd}$$

This cycle can be broken if one of the writes does not depend on the read, for example:

```
  x := y; || r := x; y := r+1-r;
```

has execution:

$$\begin{tikzcd} \boxed{\mathrm{R}\,y\,1} \arrow[d] & \boxed{\mathrm{R}\,x\,1} \\ \boxed{\mathrm{W}\,x\,1} \arrow[ur, dashed] & \boxed{\mathrm{W}\,y\,1} \arrow[ul, dashed] \end{tikzcd}$$

Note that $(\mathrm{R}\,x\,1) \not\leq (\mathrm{W}\,y\,1)$, so this does not introduce a cycle.

This model of relaxed memory is simple, and does not model many features such as coherence, non-release-acquire access, or memory fences, but it is good enough for the examples in this paper. In particular, we use the fact that relaxed memory is sensitive to data dependency to give a Spectre attack which does not depend on timing in §3.9.

## 3.7 Speculative evaluation

So far, the treatment of dependency has been fairly standard, we have included aborted events in the semantics of programs, but they have not had any impact. We shall now see that this is not always the case, that the execution of a conditional can depend on the branch not taken.

Consider the program:

```
r := z; if (r) { x := 1; } else { x := 1; }
```

Many optimizers will consider it to be the same as:

```
r := z; x := 1;
```

and so there should not be a dependency between $(\mathsf{R}\,z\,v)$ and $(\mathsf{W}\,x\,1)$. The semantics of this program is:

$$\bigcup_v (\mathsf{true} \mid \mathsf{R}\,z\,v) \longrightarrow (z \neq 0 \mid [\![\texttt{x := 1}]\!]) \sqcup (z = 0 \mid [\![\texttt{x := 1}]\!])$$

so, as in the previous section, we have executions such as:

$$\begin{tikzcd} \boxed{\mathsf{R}\,z\,1} \arrow[r] \arrow[rr, bend left] & \boxed{\mathsf{W}\,x\,1} & \boxed{\cancel{\mathsf{W}\,x\,1}} \end{tikzcd}$$

but we also have the execution:

$$\boxed{\mathsf{R}\,z\,1} \quad \boxed{\mathsf{W}\,x\,1}$$

To see why, consider $\mathcal{P}_1 \sqcup \mathcal{P}_2$ where:

$$\mathcal{P}_1 = (z \neq 0 \mid [\![\texttt{x := 1}]\!]) \quad \mathcal{P}_2 = (z = 0 \mid [\![\texttt{x := 1}]\!])$$

Now $\mathcal{P}_1$ contains the pomset $P_1$ where:

$$E_1 = \{0\} \qquad 0 \leq_1 0 \qquad \lambda_1(0) = (z \neq 0 \mid \mathsf{W}\,x\,1)$$

Similarly $\mathcal{P}_2$ contains the pomset $P_2$ where:

$$E_1 = \{0\} \qquad 0 \leq_1 0 \qquad \lambda_1(0) = (z = 0 \mid \mathsf{W}\,x\,1)$$

In particular, $E_1$ and $E_2$ overlap. One of the inhabitants of $\mathcal{P}_1 \sqcup \mathcal{P}_2$ is $P_0$ where:

$$E_0 = \{0\} \qquad 0 \leq_0 0 \qquad \lambda_0(0) = (\mathsf{true} \mid \mathsf{W}\,x\,1)$$

which is the execution we are interested in.

We can now see the counterintuitive behavior of speculation, there are programs such as `r := z; if (r) { x := 1; } else { x := 1; }` with executions in which $(\mathsf{W}\,x\,1)$ is independent of $(\mathsf{R}\,z\,1)$:

$$\boxed{\mathsf{R}\,z\,1} \quad \boxed{\mathsf{W}\,x\,1}$$

while programs such as `r := z; if (r) { x := 1; } else { y := 2; }` only have executions in which $(\mathsf{W}\,x\,1)$ is dependent on $(\mathsf{R}\,z\,1)$:

$$\begin{tikzcd} \boxed{\mathsf{R}\,z\,1} \arrow[r] \arrow[rr, bend left] & \boxed{\mathsf{W}\,x\,1} & \boxed{\cancel{\mathsf{W}\,y\,2}} \end{tikzcd}$$

so these programs have different dependency relations, depending on conditional branches that were not taken. In §3.9 we shall see that this has security implications, since relaxed memory can observe dependency. The attack is similar to Spectre, so we shall take a detour to see how Spectre can be modeled in this setting.

### 3.8 Spectre

We give a simplified model of Spectre attacks, ignoring the details of timing. In this model, we extend programs with the ability to tell whether a memory location has been touched (in practice this is implemented using timing attacks on the cache). For example, we can write a SPECTRE program as:

```
    var a;
    if (isCapability(0)) { a[SECRET] := 1; }
    else if (touched a[0]) { x := 0; }
    else if (touched a[1]) { x := 1; }
```

This is a low-security program, which is attempting to discover the value of a high-security variable SECRET. The low-security program is allowed to attempt to escalate its privileges by providing a capability which demonstrates that they are entitled to run high-security code:

```
    if (isCapability(c)) { ... escalated code ... }
    else { ... fallback code ... }
```

In this case, the `isCapability(0)` is false, so the fallback code is executed. Unfortunately, the escalated code is speculatively evaluated, which allows information to leak by testing for which memory locations have been touched.

We model the `touched` test by introducing a new read action (Why a read?) $(\mathsf{T}\,x)$ and defining:

$$[\![\mathsf{if}\ \mathsf{touched}\,x\ \mathsf{then}\,E\ \mathsf{else}\,D]\!] \quad = \quad ((\mathsf{T}\,x) \to [\![E]\!]) \sqcup [\![D]\!]$$

The additional requirement we need to add for $x$-closure is:

- if $\lambda(e) = (\phi \mid \mathsf{T}\,x)$ then there is $d \not> e$ with $\lambda(d) = (\psi \mid \mathsf{R}\,x\,v)$ or $\lambda(d) = (\psi \mid \mathsf{W}\,x\,v)$.

For example, one execution of SPECTRE is:

$$\begin{tikzcd} \cancel{\boxed{\mathsf{R}\ \mathsf{SECRET}\ 1}} \arrow[r] & \cancel{\boxed{\mathsf{W}\,a[1]\,1}} & \boxed{\mathsf{T}\,a[1]} \arrow[r] & \boxed{\mathsf{W}\,x\,1} \end{tikzcd}$$

Putting this in parallel with a high-security write to SECRET gives:

$$\begin{tikzcd} \boxed{\mathsf{W}\ \mathsf{SECRET}\ 1} \arrow[r, dashed] & \cancel{\boxed{\mathsf{R}\ \mathsf{SECRET}\ 1}} \arrow[r] & \cancel{\boxed{\mathsf{W}\,a[1]\,1}} & \boxed{\mathsf{T}\,a[1]} \arrow[r] & \boxed{\mathsf{W}\,x\,1} \end{tikzcd}$$

but due the requirement of a-closure we do *not* have:

$$\begin{tikzcd} \boxed{\mathsf{W}\ \mathsf{SECRET}\ 0} \arrow[r, dashed] & \cancel{\boxed{\mathsf{R}\ \mathsf{SECRET}\ 0}} \arrow[r] & \cancel{\boxed{\mathsf{W}\,a[1]\,0}} & \boxed{\mathsf{T}\,a[1]} \arrow[r] & \boxed{\mathsf{W}\,x\,1} \end{tikzcd}$$

Thus, the attacker has managed to leak the value of a high-security location to a low-security one.

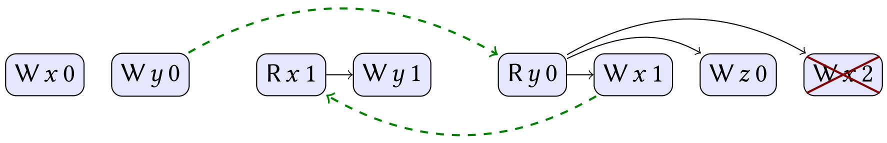
This shows how a (very abstract, untimed) model of Spectre attacks using speculative evaluation can be modeled.

### 3.9 Information flow attacks on relaxed memory

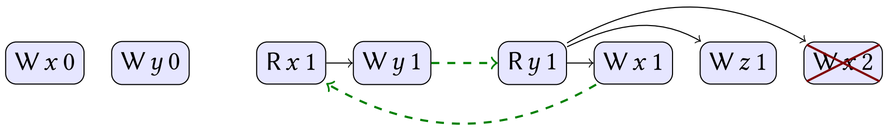
Consider an attacker program, again using security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information flows. The attacker program is:

```
  var x := 0; var y := 0;
  (
    y := x;
  ) || (
    r := y;
    if (isCapability(r)) { x := SECRET; }
    else { x := 1; z := r; }
  )
```

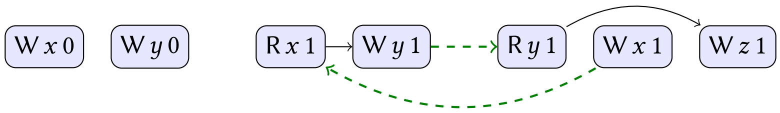
In the case where SECRET is 2, this has many executions, one of which is:



but there are no executions which exhibit (W $z$ 1), since any attempt to do so produces a cycle:



In the case where SECRET is 1, there is an execution:



Note that in this case, there is no dependency from (R $y$ 1) to (W $x$ 1), which is what makes this execution possible. Thus, if the attacker sees an execution with (W $z$ 1), they can conclude that SECRET is 1, which is an information flow attack.

This attack is not just an artifact of the model, since the same behavior can be exhibited by compiler optimizations. Consider the program fragment:

```
  r := y;
  if (isCapability(r)) { x := SECRET; }
  else { x := 1; z := r; }
```

Now, in the case where SECRET is a constant 1, the compiler can inline it:

```
  r := y;
  if (isCapability(r)) { x := 1; }
  else { x := 1; z := r; }
```

lift the assignment to x out of the if statement:

```
  r := y; x := 1;
  if (isCapability(r)) { }
  else { z := r; }
```

and then perform independent read/write reordering:

```
  x := 1; r := y;
  if (isCapability(r)) { }
  else { z := r; }
```

After these optimizations, a sequentially consistent execution exhibits (W $z$ 1).

## 3.10 Fences and release/acquire synchronization

We assume a subsets of release actions and acquire actions. Reads, writes and touches in neither release nor acquire.

We say that the action $(\mathsf{R}\,x\,v)$ reads $x$ from $v$. Likewise, $(\mathsf{R}\,x\,v)$ writes $x$ to $v$. The touch action $(\mathsf{T}\,x)$ neither reads nor writes. These notions lift to events; for example, $e$ reads $x$ from $v$ if $\lambda e$ reads $x$ from $v$.

Define $\mathrm{reads}(e) = \mathrm{reads}(a)$ when $\lambda(e) = (\phi \mid a)$. Likewise $\mathrm{writes}(e)$.

*Definition 3.1.* An *rf-pomset* is a pomset together with a relation $\mathsf{RF} \subseteq <$.

*Definition 3.2.* An rf-pomset is $x$-closed if for $e \in E$ with $\lambda(e) = (\phi \mid a)$:

- $\phi$ is independent of $x$, and
- if $e$ reads $x$ at $v$, then there is some $(d, e) \in \mathsf{RF}$ such that
  - $d$ writes $x$ at $v$ and
  - there is no $d < c < e$ such that $c$ writes $x$.

Now the general definition of prefixing:

Let $(\phi \mid a) \rightarrow \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

- $E' = E \cup \{0\}$,
- $\mathsf{RF}' = \mathsf{RF}$,
- if $d \leq e$ then $d \leq' e$,
- $\lambda'(0) = (\psi \mid a)$, where $\psi$ implies $\phi$,
- if $\lambda(e) = (\psi \mid b)$ then $\lambda'(e) = (\psi' \mid b)$ and either $\psi'$ implies $\psi$ or $0 \leq' e$ and $\psi'$ implies $\psi[\vec{v}/\vec{x}]$, where $a$ sees $\vec{x}$ at $\vec{v}$,
- if $\lambda(e) = (\psi \mid b)$ and $b$ is a release then $0 \leq' e$, and
- if $a$ is an acquire then $0 \leq' e$.

The last two conditions are new. The rest are the same as before, combining read and write.

Publication example:

```
var x; var f; x:=0; f:=0; fence; (x:=1; f_rel:=1;  ||  r:=f_acq; s:=x;)
```

We disallow the execution where `r==1` and `s!=1`

We model fences and release/acquire synchronization by introducing the following actions:

- $(\mathsf{F})$ is both a release and an acquire action
- $(\mathsf{W}_{\mathsf{rel}}\,x\,v)$ is a release action, which writes $x$ at $v$.
- $(\mathsf{R}_{\mathsf{acq}}\,x\,v)$ is an acquire action, which reads $x$ at $v$.

$$
\begin{aligned}
[\![\mathtt{fence};\,E]\!] &= (\mathsf{true} \mid \mathsf{F}) \rightarrow [\![E]\!] \\
[\![x_{\mathsf{rel}}\mathtt{:=}M;\,E]\!] &= \textstyle\bigcup_v\ (M = v \mid \mathsf{W}_{\mathsf{rel}}\,x\,v) \rightarrow [\![E]\!][M/x] \\
[\![r\mathtt{:=}x_{\mathsf{acq}};\,E]\!] &= \textstyle\bigcup_v\ (\mathsf{true} \mid \mathsf{R}_{\mathsf{acq}}\,x\,v) \rightarrow [\![E]\!][x/r]
\end{aligned}
$$

There are no additional requirements for $x$-closure is.

## 3.11 Transactions

We present a model of weakly isolated transactions. To get strong isolation, we need to make B record the reads, symmetrically to C; we also need to require in parallel composition that any order in-to/out-of a transactional event be lifted to the corresponding B/C

```
var x; var f; x:=0; f:=0; fence;
   x:=1; (begin; f:=1; f:=2; end;) || (begin; r:=f; end; s:=x;)
```

`end` ; $E$ is sugar for `if commit then` $E$ `else` $E$.

- (B) is an acquire action
- ($\mathsf{C}\,\vec{x}\,\vec{v}$) is a release action which writes $\vec{x}$ at $\vec{v}$

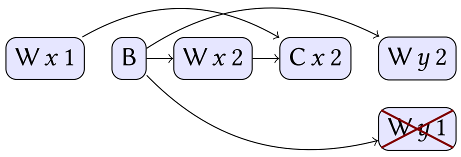
$$\begin{aligned}
[\![\texttt{begin}\,;E]\!] &= (\mathsf{true} \mid \mathsf{B}) \rightarrow [\![E]\!] \\
[\![\texttt{if}\,\texttt{commit}\,\texttt{then}\,E\,\texttt{else}\,D]\!] &= \bigcup_{\phi \text{ implies } \vec{x}=\vec{v}} ((\phi \mid \mathsf{C}\,\vec{x}\,\vec{v}) \rightarrow (\phi \mid [\![E]\!])) \sqcup (\neg\phi \mid [\![D]\!])
\end{aligned}$$

*Definition 3.3.* An rf-pomset is transaction-closed if the B and C actions with satisfiable preconditions are totally ordered by $<$.

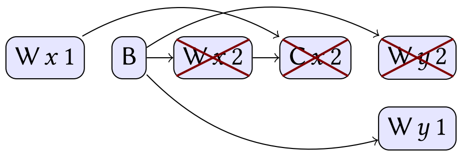
For example, the semantics of

```
x:=1; begin; x:=2; end; y:=x;
```

includes



and



## 4 EXPERIMENTS

## 5 CONCLUSIONS