Safe Optimisations for Shared-Memory Concurrent Programs

Tomer Raz

Plan

- Motivation
- Transformations
- Semantic Transformations
- Safety of Transformations
- Syntactic Transformations

Motivation

We prove that the largest classes of compiler optimisations are safe in the DRF guarantee, i.e.

- any execution of the transformed traceset has the same behavior as some execution of the original traceset, provided that the original program was data race free
- the transformations preserve data race freedom
- the transformations cannot introduce values out-of-thin-air.

Transformations

Transformations

- Trace preserving transformations
- Eliminations
- Reordering
- Memory Access Introduction

Eliminations

Thread 0	Thread 1
	`r1:=y`
`x:=2`	`print r1`
`y:=1`	`r1:=x`
`x:=1`	`r2:=x`
	`print r2`

(original)

Thread 0	Thread 1
	`r1:=y`
	`print r1`
`y:=1`	`r1:=x`
`x:=1`	`r2:=r1`
	`print r2`

(transformed)

Reorderings

Thread 0	Thread 1
`r1:=x`	`r2:=y`
`y:=r1`	`x:=1`
	`print r2`

(original)

Thread 0	Thread 1
`r1:=x`	`x:=1`
`y:=r1`	`r2:=y`
	`print r2`

(transformed)

Memory Access Introduction

```
lock m          ‖   lock m
x := 1          ‖   y := 1
print y         ‖   print x
unlock m        ‖   unlock m
```

(a) original

```
r1 := y         ‖   r2 := x
lock m          ‖   lock m
x := 1          ‖   y := 1
print y         ‖   print x
unlock m        ‖   unlock m
```

(b) with introduced reads

```
r1 := y         ‖   r2 := x
lock m          ‖   lock m
x := 1          ‖   y := 1
print r1        ‖   print r2
unlock m        ‖   unlock m
```

(c) after read elimination

Actions

- R[l=v] is a read from location l with value v
- W[l=v] a write to l with value v
- L[m] lock of monitor m
- U[m] an unlock of m
- X(v) an external action (input or output) with value v
- S(i) is a thread start action of thread i

Traces

- A sequence of memory actions of a single thread
- A program is represented as a set of traces – a traceset – with requirements:
  - Prefix closed
  - Well locked
  - Properly started

Interleaving

- Interleaving is a sequence of pairs $p = <\theta, a>$, $A(p) = a$, $T(p) = \theta$
- Interleaving I of traceset T:
  - For thread-identifier θ - the trace of θ is in T
  - $A(I_i) = S(\theta) \longrightarrow T(I_i) = \theta$
  - $A(I_i) = L[m] \longrightarrow \forall\theta \neq T(I_i)$ $|\{j \mid j < i \wedge T(I_j) = \theta \wedge A(I_j) = L[m]\}| =$

    $|\{j \mid j < i \wedge T(I_j) = \theta \wedge A(I_j) = U[m]\}|$

- Sequentially consistent interleavings of T are called *executions of T*.

Example

- Trace:
  - [S(0),R[x=v],W[y=v]]
  - [S(1),R[y=u],W[x=1],X(u)]
- Traceset:
  - Prefix closure -
    {[S(0),R[x=v],W[y=v]] | v ∈V } ∪
    {[S(1),R[y=v],W[x=1],X(v)] | v∈V}
- Interleaving:
  - [<0, S(0)>, <0, R[x=v]>, <1,S(1)>,
    <0,W[y=v]>, <1, R[y=v]>,
    <1, W[x=1]>, <1, X(v)>]

Thread 0	Thread 1
`r1:=x`	`r2:=y`
`y:=r1`	`x:=1`
	`print r2`

(original)

Thread 0	Thread 1
`r1:=x`	`x:=1`
`y:=r1`	`r2:=y`
	`print r2`

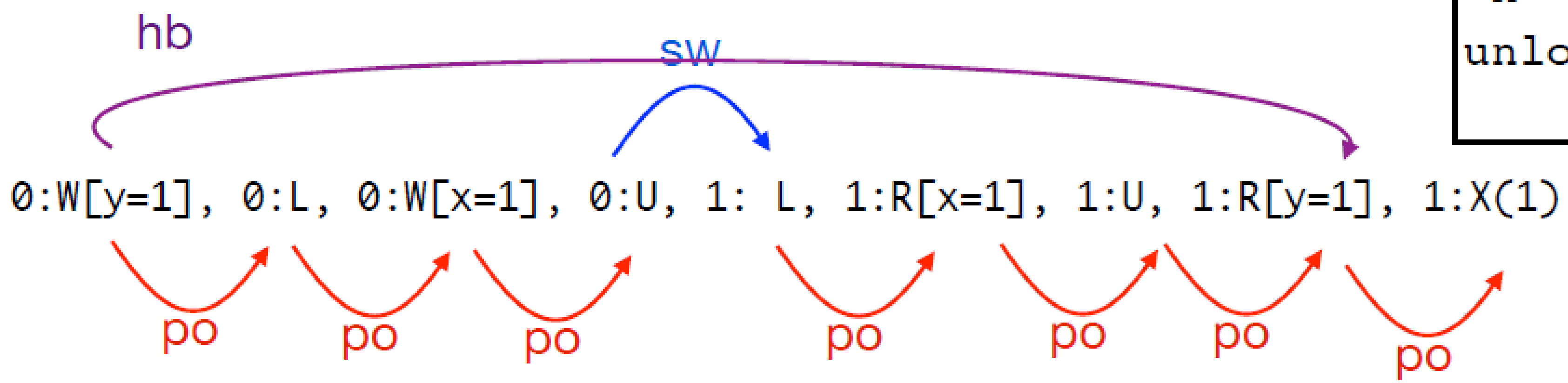
(transformed)

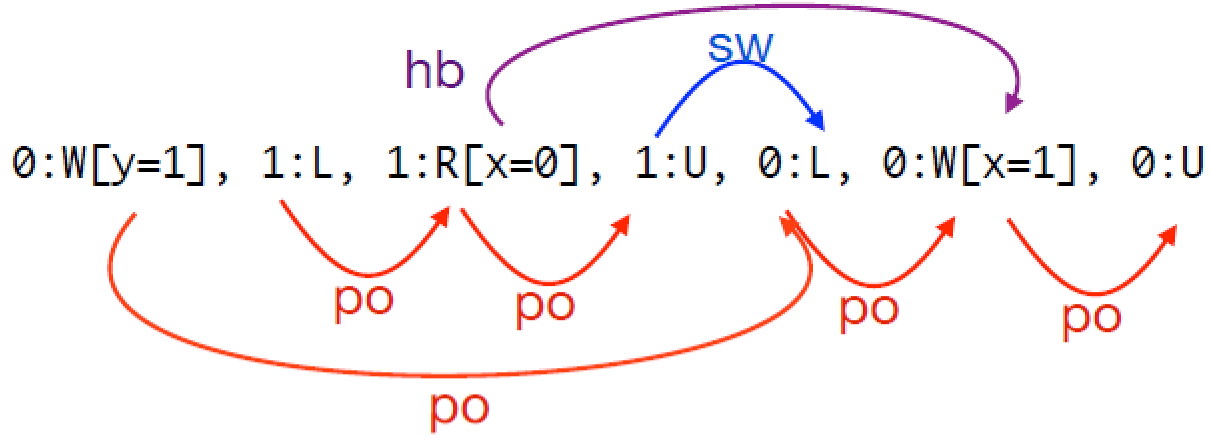
Orders

- Program order - $\leq_{\mathrm{po}}^{\mathrm{I}} = \{(\mathrm{i},\mathrm{j}) | 0 \leq \mathrm{i} \leq \mathrm{j} < |\mathrm{I}| \wedge \mathrm{T}(\mathrm{I_i}) = \mathrm{T}(\mathrm{I_j})\}$
- Synchronize with - $\leq_{\mathrm{sw}}^{\mathrm{I}} = \{(\mathrm{i},\mathrm{j}) | 0 \leq \mathrm{i} < \mathrm{j} < |\mathrm{I}| \wedge \mathrm{A}(\mathrm{I_i}), \mathrm{A}(\mathrm{I_j})\ \text{are}\ \mathit{release} - \mathit{acquire}\}$
- Happens before – transitive closure of po and sw

Example

Thread 0	Thread 1
`*y = 1`	`lock();`
`lock();`	`tmp = *x;`
`*x = 1`	`unlock();`
`unlock();`	`if tmp = 1`
	`then print *y`





Data Race Freedom

- Actions are conflicting if they access the same non-volatile location and at least one of them is a write
- An interleaving *is DRF if all conflicting accesses are ordered by the happens before relation*
- A traceset is *data race free if* none of its executions has a data race

Semantic Transformations

Semantics - Eliminations

- Wildcard trace – a trace with wildcard reads $R[x=*]$
- A wildcard trace *belongs-to traceset* T *if* T *contains all instances of the trace*
- An *instance of a wildcard interleaving is achieved by replacing each wildcard* read by a read of the same location with the value of the most recent write to the same location

Example

```
y:=1;            ║  r2:=y;
r1:=x;           ║  x:=1;
print r1;        ║
```

- Wildcard Trace:
  - [S(0),W[y=1],R[x=*]] and [S(1),R[y=*],W[x=1]]
  - [S(0),W[y=1],R[x=*],X(1)]

- Wildcard Interleaving:
  - [<0,S(0)>, <1,S(1)>, <1,R[y=0]>, <0,W[y=1]>, <1,W[x=1]>, <0,R[x=*]>, <0,X(1)>]

Eliminable

Given trace t we say that $i \in dom(t)$ (for non-volatile l, $j<i$):

- *Redundant read after read if $t_i = t_j = R[l=v]$ for some v and there is no release-acquire pair or write to l between j and i*
- *Redundant read after write if $t_i = R[l=v]$, $t_j = W[l=v]$ for some v and there is no release-acquire pair or write to l between j and i*
- *Irrelevant read if t_i is a wildcard non-volatile read*
- *Redundant write after read if $t_i = W[l=v]$, $t_j = R[l=v]$ for some v and there is no release-acquire pair or other access to l between j and i*
- *Overwritten write if $t_i = W[l=v]$, $t_j = W[l=v']$ for some v, v', and there is no release-acquire pair or other access to l between j and i*

Eliminable

Given trace t we say that $i \in dom(t)$:

- *Redundant last write if* t_i *is a normal write and there is no later release action or memory access to the same location*
- *Redundant release if* t_i *is a release and there are no later synchronization or external action*
- *Redundant external action if* t_i *is an external action and there are no later synchronization or external actions*

Example – read after read

- Original:
  - 2 cannot be printed
- Eliminated (r2:= x_{vol} turns into r2:=r1):
  - 2 can be printed

- Non volatile x doesn't solve issue, it creates DRF

Thread 0	Thread 1

r1:=x_{vol}
lock()
r2:= x_{vol}
if(r2==0)
    print y
else
    print 5
unlock()
	lock()
	x_{vol}:=1
	y:=2
	unlock()

Example – read after write

- Original:
  - 1 cannot be printed
- Eliminated (r1:=x turns into r1:=1):
  - if x:=2 is written after x:=1 then 1 is printed

Thread 0	Thread 1

lock()
x:=1
r2:=3
unlock()
lock()
r1:=x
if x==1
    print r2
else
    print r1
unlock
	lock()
	x:=2
	unlock()

- Removing release-acquire pair between the write and the read makes it so only 3 can be printed which is sound

Example – last write

- Eliminated:
  - 0 can be printed

Thread 0	Thread 1
x:=1 r1:=x print r1	y:=1

Eliminable

- *An index i is eliminable in t if i satisfies one of the conditions*
- *t' is eliminations of t if there's $S \subseteq dom(t)$ s.t. $t' = t|S$ and all indices in $dom(t) \setminus S$ are eliminable*
- *A traceset T' is an elimination* of a set of traces T *if each trace $t' \in T'$ is an elimination of some wildcard trace that belongs-to T.*

$t|S$ – subsequence of t with only indices from S

Semantics - Reorderings

- a is a non-volatile memory access, and b is a non conflicting non-volatile memory access, or an acquire action, or an external action;

- b is a non-volatile memory access, and b is a non conflicting non-volatile memory access, or an release action, or an external action

a \ b	W[x=v_x]	R[x=v_y]	Acquire	Release	External
W[y=v_y]	$x \neq y$	$x \neq y$	☑	x	☑
R[y=v_y]	$x \neq y$	☑	☑	x	☑
Acquire	x	x	x	x	x
Release	☑	☑	x	x	x
External	☑	☑	x	x	x

Example – write with write

Thread 0	Thread 1
lock() x:=1 y:=2 unlock()	lock() r2 := x unlock()

- x = y – New behavior
  - Original: [<0,S(0)><1,S(1)>, <0,L[m]>,<0,W[x=1]>,<0,W[x=2]>,<0,U[m]>,<1,L[m]>,<1,R[x=2]>,<1,U[m]>]
  - Transformed: [<0,S(0)><1,S(1)>, <0,L[m]>,<0,W[x=2]>,<0,W[x=1]>,<0,U[m]>,<1,L[m]>,<1,R[x=1]>,<1,U[m]>]
- x ≠ y – Not possible

Example – write with acquire

- Reoder write with a later acquire

Thread 0	Thread 1
x:=1	lock()
lock()	r1:=x
x:=2	unlock()
unlock()	

- After reorder: [<0,S(0)>, <0,W[x=1]>,<0,L[m]>, <0,W[x=2]>,<0,U[m]>,<1,L[m]>,<1,R[x=1]>,<1,U[m]>]

Thread 0	Thread 1
lock()	lock()
x:=1	r1:=x
x:=2	unlock()
unlock()	

Semantics - Reorderings

A traceset T' is a reordering of a traceset T if each trace t' in T' is a permutation of some trace t from T with conditions:

- Only swap reorderable actions
- Applying the permutation to any prefix of t' , that is,
  if we leave out from t all the actions that are not in the prefix, then
  the resulting trace belongs to T.

Notations

- *[a ← t. P (a)] – actions a in sequence t that satisfy condition P*
- *[f (a) | a ← t. P (a)] – [a ← t. P (a)] with each element transformed by function f*

Semantics - Reorderings

- *A bijection f is a reordering function if*

$$\mathrm{f: dom(t) \rightarrow dom(t').\, i < j, f(j) < f(i) \rightarrow t_j \text{ is reordable with } t_i}$$

- *De-permutation of a prefix of trace t*

$$f^{\rightarrow}_{<n}(t) = [t_{f^{-1}(i)} | i \leftarrow ldom(t).\, f^{-1}(i) < n]$$

- *f de-permutes t' to T if f is a reordering function t' and for* n $\leq |t'|$ -> *$f^{\rightarrow}$ (t)$\in$T*
- *set of traces T' is a reordering of a set of traces T if for each t' in T' there is a function that de-permutes t' into T*

Safety of Transformations

Safety of Transformations

- any execution of the transformed traceset has the same behavior as some execution of the original traceset, provided that the original program was data race free
- the transformations preserve data race freedom
- the transformations cannot introduce values out-of-thin-air.

Example

- We’ve seen examples of transformations breaking the first two safety constraits

- Transformation causing out of thin air value :

```
y := 42
r1 := x
if (r1 != 42) y := r1;
print r1
```

initially `x = y = 0`

~~`r1 := x`~~	`r2 := y`
~~`y := r1`~~	`x := r2`
~~`print r1`~~	`print r2`

Safety of Eliminations - Unelimination

Matching f $\mathrm{f: dom(I) \rightarrow dom(I') \; s.t. \; I_i = I'_{f(i)}}$ where $dom(t) = \{0, \ldots, |t| - 1\}$

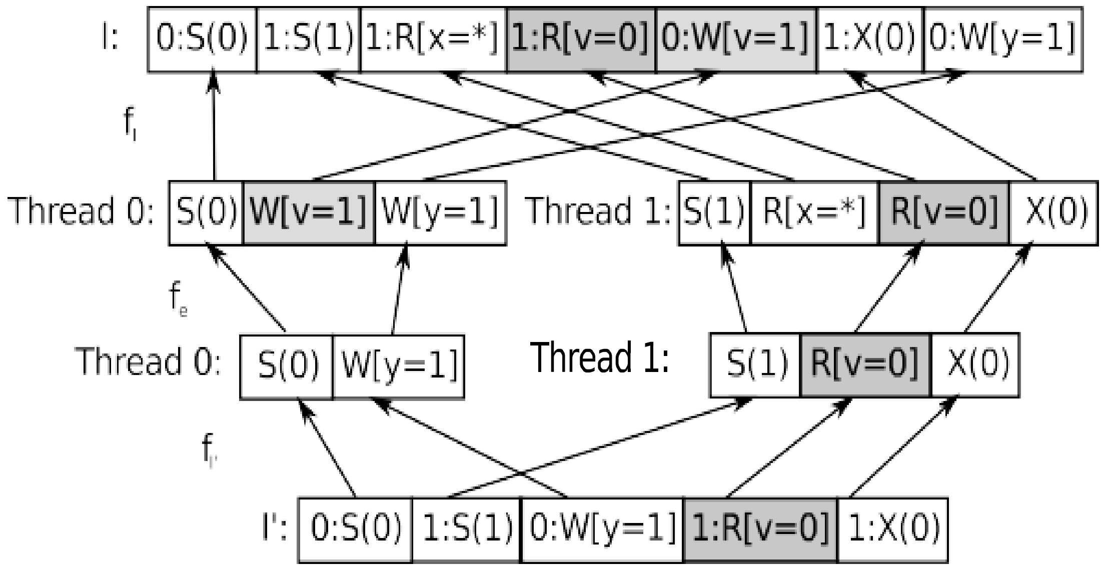
Unelimination function is a complete matching between I, I' s.t.:

- $i < j \in dom(I') \wedge T(I_i') = T(I_j') \rightarrow f(i) < f(j)$
- $i < j \in dom(I') \wedge A(I_i'), A(I_j')\ are\ synchronization\ or\ external\ actions$
  $\rightarrow f(i) < f(j)$
- $i \in range(f), j \in dom(I) \backslash range(f)$
  $\wedge A(I_i), A(I_j)\ are\ synchronization\ or\ external\ actions\ \rightarrow i < j$
- $i \in dom(I) \backslash range(f) \rightarrow is\ eliminable$

Unelimination

- *Let traceset* T' *be an elimination of traceset* T *and* I' *an interleaving of* T'. *Then there is a wildcard interleaving* I *belonging-to* T *and an unelimination function* f *from* I' *to* I
- *Let traceset* T' *be an elimination of a data free traceset* T. *Then* T' *is data race free and any execution of* T' *has the same behaviour as some execution of* T.

Unelimination Example



Unelimination construction

Safety of Reorderings

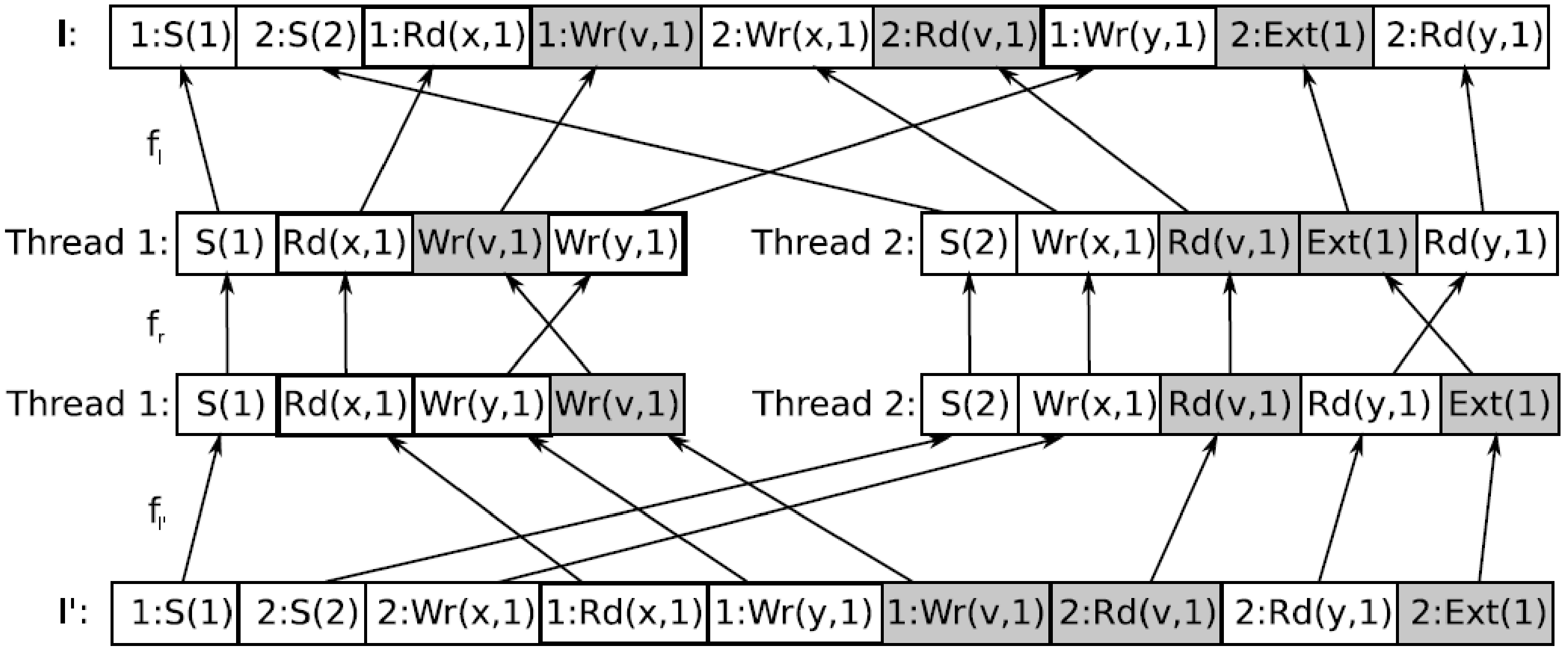
Unordering function is a complete matching between I, I' s.t.:

- $i < j \in dom(I') \wedge T(I_i') = T(I_j') \wedge A(I_i'), A(I_j')\ not\ reordable \rightarrow f(i) < f(j)$
- $i < j \in dom(I') \wedge A(I_i'), A(I_j')\ are\ synchronization\ or\ external\ actions \rightarrow f(i) < f(j)$
- $for\ each\ thread\ \theta, the\ permutation\ f\ restricted\ to\ actions\ of\ \theta\ de - permutes\ the\ trace\ of\ \theta\ in\ I'\ into\ T$

Unordering

Suppose that traceset T' is a reordering of a data race free traceset T. Then any execution of T' has the same behaviour as some execution of T. Moreover, T' is data race free.

Unordering Example



Unordering construction

Out-of-thin-air

- Origin – *t is an origin for v if there is $i \in dom(t)$ s.t. t_i is a write of v or an external action with value v and there is no $j<i$ s.t. t_j is a read of v*
- *Let traceset T' be a reordering or an elimination of traceset T and suppose that no trace in T is an origin for v, then no trace in T' is an origin for v.*
- *If T does not contain an origin for a value, no execution of T can output that value*

Syntactic Transformations

The syntax

$$
\begin{aligned}
ri &::= \; r \mid i \\
T &::= \; ri \; \texttt{==} \; ri \mid ri \; \texttt{!=} \; ri \\
S &::= \; l \; \texttt{:=} \; r\texttt{;} \mid r \; \texttt{:=} \; l\texttt{;} \mid r \; \texttt{:=} \; ri\texttt{;} \mid \texttt{lock} \; m\texttt{;} \mid \texttt{unlock} \; m\texttt{;} \mid \texttt{skip;} \\
&\quad \mid \texttt{print} \; r\texttt{;} \mid \texttt{\{}L\texttt{\}} \mid \texttt{if} \; (T) \; S \; \texttt{else} \; S \mid \texttt{while} \; (T) \; S \\
L &::= \; S \mid S \, L \\
P &::= \; L \,||\, L \,||\, \ldots \,||\, L
\end{aligned}
$$

A simple concurrent language – syntax.

Notations

- A thread local configuration is $<\Lambda, \sigma, C>$:
  - Λ *is a function that maps monitor names to the nesting level of locks*
  - *local state* σ *maps register names to values*
  - C *is a code fragment, which is either* S *or* L *or* P *from the syntax*
- The step relation $<\Lambda, \sigma, C> \rightarrow^{a} <\Lambda', \sigma', C'>$ for action a
- $<\Lambda, \sigma, C> \Rightarrow^{t}_{n} <\Lambda', \sigma', C'>$ for a sequence of n *transitions*
- $<\Lambda, \sigma, C> \Downarrow t$ if there exists $<\Lambda', \sigma', C'>$ $s.t.$ $<\Lambda, \sigma, C> \Rightarrow^{t}_{n}$ $<\Lambda', \sigma', C'>$
- $[c]_{\Lambda,\sigma} = \{t \mid <\Lambda, \sigma, C> \Downarrow t\}$

$$\begin{array}{llll}
\langle \Lambda, \sigma, r\texttt{:=}ri\texttt{;}\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma[r \mapsto \mathsf{Val}(\sigma, ri)], \texttt{skip;}\rangle & & (\textsc{Regs}) \\
\langle \Lambda, \sigma, x\texttt{:=}r\texttt{;}\rangle & \xrightarrow{\mathrm{W}[x=\sigma(r)]} \langle \Lambda, \sigma, \texttt{skip;}\rangle & & (\textsc{Write}) \\
\langle \Lambda, \sigma, r\texttt{:=}x\texttt{;}\rangle & \xrightarrow{\mathrm{R}[x=v]} \langle \Lambda, \sigma[r \mapsto v], \texttt{skip;}\rangle & \text{where } v \in \tau(x) & (\textsc{Read}) \\
\langle \Lambda, \sigma, \texttt{lock}\ m\texttt{;}\rangle & \xrightarrow{\mathrm{L}[m]} \langle \Lambda[m \mapsto \Lambda(m)+1], \sigma, \texttt{skip;}\rangle & & (\textsc{Lock}) \\
\langle \Lambda, \sigma, \texttt{unlock}\ m\texttt{;}\rangle & \xrightarrow{\mathrm{U}[m]} \langle \Lambda[m \mapsto \Lambda(m)-1], \sigma, \texttt{skip;}\rangle & \text{where } \Lambda(m) > 0 & (\textsc{Ulk}) \\
\langle \Lambda, \sigma, \texttt{unlock}\ m\texttt{;}\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, \texttt{skip;}\rangle & \text{where } \Lambda(m) = 0 & (\textsc{E-Ulk}) \\
\langle \Lambda, \sigma, \texttt{print}\ r\texttt{;}\rangle & \xrightarrow{\mathrm{X}(\sigma(r))} \langle \Lambda, \sigma, \texttt{skip;}\rangle & & (\textsc{Ext}) \\
\langle \Lambda, \sigma, \texttt{if (}T\texttt{)}\ S_1\ \texttt{else}\ S_2\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, S_1\rangle & \text{if } \mathsf{Val}(\sigma, T) = \mathbf{tt} & (\textsc{Cond-T}) \\
\langle \Lambda, \sigma, \texttt{if (}T\texttt{)}\ S_1\ \texttt{else}\ S_2\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, S_2\rangle & \text{if } \mathsf{Val}(\sigma, T) = \mathbf{ff} & (\textsc{Cond-F}) \\
\langle \Lambda, \sigma, \texttt{while (}T\texttt{)}\ S\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, S\texttt{;while (}T\texttt{)}\ S\rangle & \text{if } \mathsf{Val}(\sigma, T) = \mathbf{tt} & (\textsc{Loop-T}) \\
\langle \Lambda, \sigma, \texttt{while (}T\texttt{)}\ S\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, \texttt{skip;}\rangle & \text{if } \mathsf{Val}(\sigma, T) = \mathbf{ff} & (\textsc{Loop-F}) \\
\langle \Lambda, \sigma, \texttt{skip;}L\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, L\rangle & & (\textsc{Seq}) \\
\langle \Lambda, \sigma, \texttt{\{skip;\}}\rangle & \xrightarrow{\tau} \langle \Lambda, \sigma, \texttt{skip;}\rangle & & (\textsc{Block}) \\
\langle \Lambda, \sigma, L_0 \mathbin{||} \dots \mathbin{||} L_n\rangle & \xrightarrow{\mathrm{S}(i)} \langle \Lambda, \sigma, L_i\rangle & \text{where } 0 \le i \le n & (\textsc{Par})
\end{array}$$

$$\frac{\langle \Lambda, \sigma, S\rangle \xrightarrow{a} \langle \Lambda', \sigma', S'\rangle}{\langle \Lambda, \sigma, S\ L\rangle \xrightarrow{a} \langle \Lambda', \sigma', S' L\rangle}(\textsc{Ev-Seq}) \qquad \frac{\Lambda, \sigma, L \xrightarrow{a} \Lambda', \sigma', L'}{\langle \Lambda, \sigma, \{L\}\rangle \xrightarrow{a} \langle \Lambda', \sigma', \{L'\}\rangle}(\textsc{Ev-Block})$$

Figure 7. Small-step Trace Semantics.

$$\frac{}{\langle \Lambda, \sigma, C\rangle \overset{[]}{\underset{0}{\Longrightarrow}} \langle \Lambda, \sigma, C\rangle}(\textsc{Tr-Id}) \qquad \frac{\langle \Lambda, \sigma, C\rangle \xrightarrow{\tau} \langle \Lambda'', \sigma'', C''\rangle \quad \langle \Lambda'', \sigma'', C''\rangle \overset{\alpha}{\underset{n}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}{\langle \Lambda, \sigma, C\rangle \overset{\alpha}{\underset{n+1}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}(\textsc{Tr-SeqT})$$

$$\frac{\langle \Lambda, \sigma, C\rangle \xrightarrow{a} \langle \Lambda'', \sigma'', C''\rangle \quad a \neq \tau \quad \langle \Lambda'', \sigma'', C''\rangle \overset{\alpha}{\underset{n}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}{\langle \Lambda, \sigma, C\rangle \overset{a::\alpha}{\underset{n+1}{\Longrightarrow}} \langle \Lambda', \sigma', C'\rangle}(\textsc{Tr-SeqA})$$

Figure 8. Multi-step Trace Semantics.

$$\frac{}{S \overset{t}{\leadsto} S}(\text{T-Id}) \qquad \frac{L \overset{t}{\leadsto} L'}{\{L\} \overset{t}{\leadsto} \{L'\}}(\text{T-Block}) \qquad \frac{S_1 \overset{t}{\leadsto} S_1' \quad L_2 \overset{t}{\leadsto} L_2'}{S_1\ L_2 \overset{t}{\leadsto} S_1'\ L_2'}(\text{T-Seq})$$

$$\frac{S_1 \overset{t}{\leadsto} S_1' \quad S_2 \overset{t}{\leadsto} S_2'}{\texttt{if}\ (T)\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \overset{t}{\leadsto} \texttt{if}\ (T)\ \texttt{then}\ S_1'\ \texttt{else}\ S_2'}(\text{T-If})$$

$$\frac{S \overset{t}{\leadsto} S'}{\texttt{while}\ (T)\ S \overset{t}{\leadsto} \texttt{while}\ (T)\ S'}(\text{T-While}) \qquad \frac{\forall i \in \{0, \ldots, n\}.\ S_i \overset{t}{\leadsto} S_i'}{S_0 \mid\mid \ldots \mid\mid S_n \overset{t}{\leadsto} S_0' \mid\mid \ldots \mid\mid S_n'}(\text{T-Par})$$

Figure 9. Transformation template.

Sytactic Eliminations

Lemma 4. *Let C be a code fragment and $C \overset{e}{\rightsquigarrow} C'$. Then for any monitor states Λ, Λ', register states σ, σ' and trace t' we have:*

- *If $\langle \Lambda, \sigma, C' \rangle \Downarrow t'$ then there is a wildcard trace t such that $t \rightarrow_e t'$ and for any instance $\hat{t}$ of t we have $\langle \Lambda, \sigma, C \rangle \Downarrow \hat{t}$.*
- *If $\langle \Lambda, \sigma, C' \rangle \overset{t'}{\Longrightarrow} \langle \Lambda', \sigma', \texttt{skip;} \rangle$ then there is a wildcard trace t such that $t \rightarrow_e t'$ and for any instance $\hat{t}$ of t we have $\langle \Lambda, \sigma, C \rangle \overset{\hat{t}}{\Longrightarrow} \langle \Lambda', \sigma', \texttt{skip;} \rangle$.*

Theorem 3. *Suppose that $P \overset{e}{\rightsquigarrow} P'$ and $[\![P]\!]$ is data race free. Then $[\![P']\!]$ is data race free, and any execution of $[\![P']\!]$ has the same behaviour as some execution of $[\![P]\!]$.*

$$\frac{x \text{ not volatile} \quad r_1, r_2, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{r_1 \texttt{:=} x\texttt{;}\ \ S\texttt{;}\ \ r_2 \texttt{:=} x \ \overset{e}{\rightsquigarrow}\ r_1 \texttt{:=} x\texttt{;}\ \ S\texttt{;}\ \ r_2 \texttt{:=} r_1}\text{(E-RAR)}$$

$$\frac{x \text{ not volatile} \quad r_1, r_2, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{x \texttt{:=} r_1\texttt{;}\ \ S\texttt{;}\ \ r_2 \texttt{:=} x \ \overset{e}{\rightsquigarrow}\ x \texttt{:=} r_1\texttt{;}\ \ S\texttt{;}\ \ r_2 \texttt{:=} r_1}\text{(E-RAW)}$$

$$\frac{x \text{ not volatile} \quad r, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{r \texttt{:=} x\texttt{;}\ \ S\texttt{;}\ \ x \texttt{:=} r \ \overset{e}{\rightsquigarrow}\ r \texttt{:=} x\texttt{;}\ \ S\texttt{;}}\text{(E-WAR)}$$

$$\frac{x \text{ not volatile} \quad r_1, r_2, x \notin \mathrm{fv}(S) \quad S \text{ sync-free}}{x \texttt{:=} r_1\texttt{;}\ \ S\texttt{;}\ \ x \texttt{:=} r_2 \ \overset{e}{\rightsquigarrow}\ S\texttt{;}\ \ x \texttt{:=} r_2}\text{(E-WBW)}$$

$$\frac{x \text{ not volatile}}{r \texttt{:=} x\texttt{;}\ \ r \texttt{:=} i \ \overset{e}{\rightsquigarrow}\ r \texttt{:=} i}\text{(E-IR)}$$

Figure 10. Additional rules for syntactic elimination.

Syntactic Reorderings

Lemma 5. *Assume that* $C \overset{r}{\rightsquigarrow} C'$. *Then for each* Λ *and* σ *there is a prefix closed set of traces* T *satisfying these conditions: (i) the set of traces* $[\![C]\!]_{\Lambda,\sigma}$ *is a subset of* T, *(ii) each trace from* T *is an elimination of some wildcard trace that belongs-to* $[\![C]\!]_{\Lambda,\sigma}$, *(iii) for each trace* t', *if* $\langle\Lambda,\sigma,C'\rangle \Downarrow t'$ *holds then there is a function that*

Theorem 4. *Suppose that* $P \overset{r}{\rightsquigarrow} P'$ *and* $[\![P]\!]$ *is data race free. Then* $[\![P']\!]$ *is data race free, and any execution of* $[\![P']\!]$ *has the same behaviour as some execution of* $[\![P]\!]$.

$$\frac{r_1 \neq r_2 \qquad x \text{ not volatile}}{r_1\texttt{:=}x\texttt{;}\ r_2\texttt{:=}y\texttt{;}\ \overset{r}{\rightsquigarrow}\ r_2\texttt{:=}y\texttt{;}\ r_1\texttt{:=}x\texttt{;}}\text{(R-RR)}$$

$$\frac{x \neq y \qquad y \text{ not volatile}}{x\texttt{:=}r_1\texttt{;}\ y\texttt{:=}r_2\texttt{;}\ \overset{r}{\rightsquigarrow}\ y\texttt{:=}r_2\texttt{;}\ x\texttt{:=}r_1\texttt{;}}\text{(R-WW)}$$

$$\frac{r_1 \neq r_2 \qquad x \neq y \qquad x \text{ or } y \text{ not volatile}}{x\texttt{:=}r_1\texttt{;}\ r_2\texttt{:=}y\texttt{;}\ \overset{r}{\rightsquigarrow}\ r_2\texttt{:=}y\texttt{;}\ x\texttt{:=}r_1\texttt{;}}\text{(R-WR)}$$

$$\frac{r_1 \neq r_2 \qquad x \neq y \qquad x, y \text{ not volatile}}{r_1\texttt{:=}x\texttt{;}\ y\texttt{:=}r_2\texttt{;}\ \overset{r}{\rightsquigarrow}\ y\texttt{:=}r_2\texttt{;}\ r_1\texttt{:=}x\texttt{;;}}\text{(R-RW)}$$

$$\frac{x \text{ not volatile}}{x\texttt{:=}r\texttt{;}\ \texttt{lock}\ m\texttt{;}\ \overset{r}{\rightsquigarrow}\ \texttt{lock}\ m\texttt{;}\ x\texttt{:=}r\texttt{;}}\text{(R-WL)}$$

$$\frac{x \text{ not volatile}}{r\texttt{:=}x\texttt{;}\ \texttt{lock}\ m\texttt{;}\ \overset{r}{\rightsquigarrow}\ \texttt{lock}\ m\texttt{;}\ r\texttt{:=}x\texttt{;}}\text{(R-RL)}$$

$$\frac{x \text{ not volatile}}{\texttt{unlock}\ m\texttt{;}x\texttt{:=}r\texttt{;}\ \overset{r}{\rightsquigarrow}\ x\texttt{:=}r\texttt{;unlock}\ m\texttt{;}}\text{(R-UW)}$$

$$\frac{x \text{ not volatile}}{\texttt{unlock}\ m\texttt{;}r\texttt{:=}x\texttt{;}\ \overset{r}{\rightsquigarrow}\ r\texttt{:=}x\texttt{;unlock}\ m\texttt{;}}\text{(R-UR)}$$

$$\frac{r_1 \neq r_2 \qquad x \text{ not volatile}}{\texttt{print}\ r_1\texttt{;}r_2\texttt{:=}x\texttt{;}\ \overset{r}{\leftrightsquigarrow}\ r_2\texttt{:=}x\texttt{;print}\ r_1\texttt{;}}\text{(R-XR)}$$

$$\frac{x \text{ not volatile}}{\texttt{print}\ r_1\texttt{;}x\texttt{:=}r_2\texttt{;}\ \overset{r}{\leftrightsquigarrow}\ x\texttt{:=}r_2\texttt{;print}\ r_1\texttt{;}}\text{(R-XW)}$$

Figure 11. Additional rules for syntactic reordering.

Out-of-thin-air

Lemma 6. *Let v be a value such that v is not a default value for any location, i.e., $v \neq 0$. Let P be a program without any statement of the form $r := v$, where r is a register name. Then no trace in the traceset of P is an origin for the value v.*

Theorem 5. *Suppose that c is a constant different from 0, and P a program that does not contain a statement of the form $r := c$, where r is a register. Let P' be a program obtained from P by any composition of syntactic reorderings or eliminations. Then P' cannot output c.*