Simuliris

A Separation Logic Framework for Verifying Concurrent Program Optimizations

Lennard Gähler, Michael Sammler, Simon Spies, Ralf Jung,
Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, Derek Dreyer
```c
int mult(int *x, int *y) {
    int i = 0; int sum = *y;
    while (i != *x - 1) {
        i += 1; sum += *y;
    }
    return sum;
}

int opt(int *x, int *y) {
    int n = *x;
    int m = *y;
    int i = 0; int sum = m;
    while (i != n - 1) {
        i += 1; sum += m;
    }
    return sum;
}
```

implements multiplication of positive integers
int mult(int *x, int *y) {
    int i = 0; int sum = *y;
    while (i != *x - 1) {
        i += 1; sum += *y;
    }
    return sum;
}

int opt(int *x, int *y) {
    int n = *x;
    int m = *y;
    int i = 0; int sum = m;
    while (i != n - 1) {
        i += 1; sum += m;
    }
    return sum;
}

implements multiplication of positive integers

How can we prove correctness of these program optimizations?
Can concurrent writes break the optimization?

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```
Can concurrent writes break the optimization?

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

\[
\begin{align*}
\text{\texttt{*x} } &= \text{ 2; } \\
\text{\texttt{*y} } &= \text{ 42; }
\end{align*}
\]
Can concurrent writes break the optimization?

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

→

```
*x = 2;
*y = 42;
```

can produce results not possible for the unoptimized program!
Can concurrent writes break the optimization?

current state: *x *y

1 99

optimized program:

→int n = *x;
  int m = *y;
  int i = 0; int sum = m;
  while (i != n - 1) {
    i += 1; sum += m;
  }
  return sum;

→*x = 2;
  *y = 42;
Can concurrent writes break the optimization?

current state: \( x \quad y \quad n \)

\[
\begin{array}{cccc}
1 & 99 & 1 \\
\end{array}
\]

optimized program:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

\[ \rightarrow x = 2; \quad \rightarrow y = 42; \]
Can concurrent writes break the optimization?

current state: \(*x\) \(*y\) \(n\)

\[
\begin{array}{ccc}
2 & 99 & 1 \\
\end{array}
\]

optimized program:

\[
\begin{align*}
\text{int } n &= \ast x; \\
\text{int } m &= \ast y; \\
\text{int } i &= 0; \text{int sum } &= m; \\
\text{while } (i \neq n - 1) \{ \\
\quad i &= i + 1; \text{sum } += m; \\
\} \\
\text{return sum;}
\end{align*}
\]

\[
\begin{array}{c}
\ast x = 2; \\
\ast y = 42;
\end{array}
\]
Can concurrent writes break the optimization?

current state:  

*\( x \)  \*\( y \)  \( n \)  

2  42  1

optimized program:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
  i += 1; sum += m;
}
return sum;
```

\( \rightarrow \)

\( \star x = 2; \)

\( \star y = 42; \)
Can concurrent writes break the optimization?

current state: \( *x \quad *y \quad n \quad m \)

\[
\begin{array}{llll}
2 & 42 & 1 & 42 \\
\end{array}
\]

optimized program:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

```c
* x = 2;
* y = 42;
```

The optimized program can produce the result 42 with initial \( *x = 1 \) and \( *y = 99 \) (by using the old value 1 of \( x \) and the new value 42 of \( y \))
Can concurrent writes break the optimization?

current state: \[*x\] \[*y\] \[n\] \[m\] \[sum\]

\[2\] \[42\] \[1\] \[42\] \[42\]

optimized program:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

The optimized program can produce the result 42 with initial \[*x* = 1 and \[*y* = 99
(by using the old value 1 of \(x\) and the new value 42 of \(y\))
The unoptimized program cannot produce the result 42

current state: 

\[
\begin{array}{cc}
  *x & *y \\
  1 & 99 \\
\end{array}
\]

unoptimized program:

```c
int i = 0;
int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
} 
return sum;
```

The unoptimized program can also not produce the result 42 with initial \( *x = 1 \) and \( *y = 99 \) (if the new value 42 of \( y \) is read, also the new value 2 of \( x \) is read).
The unoptimized program cannot produce the result 42

current state: | *x | *y |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

unoptimized program:

```c
int i = 0;
int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

```c
* x = 2;
* y = 42;
```
The unoptimized program cannot produce the result 42

current state:  

\[
\begin{array}{c|c}
*x & *y \\
2 & 42 \\
\end{array}
\]

unoptimized program:

\[
\begin{align*}
\text{int } i &= 0; \\
\text{int sum } &= *y; \\
\text{while } (i \neq *x - 1) \\
&\text{ \quad \{ } \\
&\quad i += 1; \text{ sum } += *y; \\
&\quad \} \\
\text{return sum};
\end{align*}
\]

\[
\begin{array}{c|c}
*x & *y \\
2 & 42 \\
\end{array}
\]
The unoptimized program cannot produce the result 42

current state: \*x \*y

2 42

unoptimized program:

```
int i = 0;
int sum = \*y;
while (i != \*x - 1) {
    i += 1; sum += \*y;
}
return sum;
```

\*x = 2;
\*y = 42;
The unoptimized program cannot produce the result 42

current state:          *x  *y
                       2   42

unoptimized program:

```c
int i = 0;
int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

The unoptimized program can **not** produce the result 42

with initial *x = 1 and *y = 99

(if the new value 42 of y is read, also the new value 2 of x is read)
The unoptimized program cannot produce the result 42

current state:  

\[ \begin{array}{cc} *x & *y \\ 2 & 42 \end{array} \]

unoptimized program:

```c
int i = 0;
int sum = *y;
while (i != -1) {
    i += 1;
    sum += *y;
}
return sum;
```

The optimization seems to introduce new program behavior!

The unoptimized program can not produce the result 42 with initial \( *x = 1 \) and \( *y = 99 \) (if the new value 42 of \( y \) is read, also the new value 2 of \( x \) is read)
Correctness under concurrency: relying on undefined behavior

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Data races are undefined behavior (UB) in C/C++/unsafe Rust.
The compiler may assume their absence.
Correctness under concurrency: relying on undefined behavior

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

*Data races* are *undefined behavior (UB)* in C/C++/unsafe Rust.
Correctness under concurrency: relying on undefined behavior

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

Data races are undefined behavior (UB) in C/C++/unsafe Rust. The compiler may assume their absence.
Correctness under concurrency: relying on undefined behavior

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

The optimization is correct. But how can we prove that?

Data races are undefined behavior (UB) in C/C++/unsafe Rust. The compiler may assume their absence.
The problem

How can we prove correctness of concurrent program optimizations relying on data race UB and involving loops?

<table>
<thead>
<tr>
<th></th>
<th>data race UB</th>
<th>concurrent</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Ševčík, 2009], [Morisset et al., 2013]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>[Vafeiadis et al., 2015]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>CAS/Concurrent CompCert</td>
<td>~</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>CompCertTSO [Ševčík et al., 2013]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CCAL (CompCertX) [Gu et al., 2018]</td>
<td>x</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>[Liang and Feng, 2016]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ReLoC [Frumin et al., 2018]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Tassarotti et al., 2017]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfinite Iris [Spies et al., 2021]</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Stacked Borrows [Jung et al., 2020]</td>
<td>x</td>
<td>x</td>
<td>~</td>
</tr>
</tbody>
</table>
The problem

How can we prove correctness of concurrent program optimizations relying on data race UB and involving loops?

<table>
<thead>
<tr>
<th></th>
<th>data race UB</th>
<th>concurrent</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Ševčík, 2009], [Morisset et al., 2013]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>[Vafeiadis et al., 2015]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>CAS/Concurrent CompCert</td>
<td>~</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>CompCertTSO [Ševčík et al., 2013]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CCAL (CompCertX) [Gu et al., 2018]</td>
<td>x</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>[Liang and Feng, 2016]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ReLoC [Frumin et al., 2018]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Tassarotti et al., 2017]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfinite Iris [Spies et al., 2021]</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Stacked Borrows [Jung et al., 2020]</td>
<td>x</td>
<td>x</td>
<td>~</td>
</tr>
</tbody>
</table>
The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

<table>
<thead>
<tr>
<th></th>
<th>data race UB</th>
<th>concurrent</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Ševčík, 2009], [Morisset et al., 2013]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>[Vafeiadis et al., 2015]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>CAS/Concurrent CompCert</td>
<td>~</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>CompCertTSO [Ševčík et al., 2013]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Liang and Feng, 2016]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ReLoC [Frumin et al., 2018]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Tassarotti et al., 2017]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfinite Iris [Spies et al., 2021]</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Stacked Borrows [Jung et al., 2020]</td>
<td>✗</td>
<td>✗</td>
<td>~</td>
</tr>
</tbody>
</table>

- **can only handle finite traces**
- **cannot handle potentially unbounded loops**
The problem

How can we prove correctness of **concurrent** program optimizations relying on **data race UB** and involving **loops**?

<table>
<thead>
<tr>
<th></th>
<th>data race UB</th>
<th>concurrent</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Ševčík, 2009], [Morisset et al., 2013]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>[Vafeiadis et al., 2015]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>CAS/Concurrent CompCert</td>
<td>~</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>CompCertTSO [Ševčík et al., 2013]</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Tassarotti et al., 2017]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfinite Iris [Spies et al., 2021]</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Stacked Borrows [Jung et al., 2020]</td>
<td>x</td>
<td>x</td>
<td>~</td>
</tr>
</tbody>
</table>

- no optimizations involving synchronizing operations (e.g., atomic reads)
The problem

How can we prove correctness of concurrent program optimizations relying on data race UB and involving loops?

<table>
<thead>
<tr>
<th>Our approach</th>
<th>data race UB</th>
<th>concurrent</th>
<th>loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Ševčík, 2009], [Morisset et al., 2013]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Vafeiadis et al., 2015]</td>
<td>✓</td>
<td>✓</td>
<td>~</td>
</tr>
<tr>
<td>CAS/Concurrent CompCert</td>
<td>~</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>CompCertTSO [Ševčík et al., 2013]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CCAL (CompCertX) [Gu et al., 2018]</td>
<td>✗</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>[Liang and Feng, 2016]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ReLoC [Frumin et al., 2018]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Tassarotti et al., 2017]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfinite Iris [Spies et al., 2021]</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Stacked Borrows [Jung et al., 2020]</td>
<td>✗</td>
<td>✗</td>
<td>~</td>
</tr>
</tbody>
</table>
Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```
Key idea: ownership acquisition on unsynchronized accesses

unoptimized:
reach unsynchronized access to $y$

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```
Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

reach unsynchronized access to \( y \)

\[
\begin{align*}
\text{int } i &= 0; \text{ int sum } = *y; \\
\text{while } (i \neq *x - 1) \{ \\
&\quad i += 1; \text{ sum } += *y; \\
\} \\
\text{return sum;}
\end{align*}
\]

optimized:

obtain ownership with proof rule

\[
\begin{align*}
\{y \mapsto_{\text{src}} z_y \ast y \mapsto_{\text{tgt}} z_y\} \Rightarrow \\
\text{int } n = *x; \\
\text{int } m = *y; \\
\text{int } i = 0; \text{ int sum } = m; \\
\text{while } (i \neq n - 1) \{ \\
&\quad i += 1; \text{ sum } += m; \\
\} \\
\text{return sum;}
\end{align*}
\]
Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

```
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

reach unsynchronized access to x

obtain ownership with proof rule
Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

reach unsynchronized access to $x$

$\{ y \mapsto_{src} z_y * y \mapsto_{tgt} z_y * x \mapsto_{src} z_x * x \mapsto_{tgt} z_x \} \triangleright$

int $i = 0$; int sum = *y;
while ($i != *x - 1$) {
    $i += 1$; sum += *y;
}

return sum;

optimized:

obtain ownership with proof rule

int $n = *x$;
int $m = *y$;

int $i = 0$; int sum = $m$;
while ($i != n - 1$) {
    $i += 1$; sum += $m$;
}

return sum;
Key idea: ownership acquisition on unsynchronized accesses

unoptimized:

```c
int i = 0; int sum = *y;
while (i != *x - 1) {
    i += 1; sum += *y;
}
return sum;
```

optimized:

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n - 1) {
    i += 1; sum += m;
}
return sum;
```

obtain ownership
with proof rule

retain ownership
troughout the loop
Simuliris: a separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...
Simuliris: a separation logic-based simulation framework

- logic for data race based optimizations
- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

- Stacked Borrows for Rust [Jung et al., 2020] + concurrency

Simuliris: separation logic-based simulation framework
| logic for data race based optimizations | Stacked Borrows for Rust [Jung et al., 2020] + concurrency |

**Simuliris**: separation logic-based simulation framework
- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

Fully mechanized in the Coq proof assistant based on the Iris framework
Simuliris: a separation logic-based simulation framework

- logic for data race based optimizations

Simuliris: separation logic-based simulation framework
- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

fully mechanized in the Coq proof assistant based on the Iris framework

Stacked Borrows for Rust [Jung et al., 2020] + concurrency
Simuliris: a separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

fully mechanized in the Coq proof assistant based on the Iris framework

Stacked Borrows for Rust [Jung et al., 2020] + concurrency
Key ingredient: a powerful simulation relation

traditional coinductive simulation + modern separation logic
Key ingredient: a powerful simulation relation

\{P\} \quad e_s \preceq e_t \quad \{v_s, v_t.\ Q\}

source expression (unoptimized) \quad target expression (optimized)

coinductive simulation \quad separation logic
Key ingredient: a powerful simulation relation

\[
\{P\} \; e_s \; \preceq \; e_t \; \{v_s, v_t. \; Q\}
\]

source expression  
(unoptimized)  

\[ \rightarrow \]

\[ \rightarrow \]

target expression  
(optimized)  

coinductive simulation  separation logic  

• coinduction  
• reasoning about UB  
• flexible stuttering
Key ingredient: a powerful simulation relation

\[ \{ P \} \ e_s \preceq e_t \ \{ v_s, v_t. \ Q \} \]

source expression (unoptimized) \hspace{1cm} target expression (optimized)

coinductive simulation
- coinduction
- reasoning about UB
- flexible stuttering

separation logic
- compositional proof rules
Key ingredient: a powerful simulation relation

\[ \{P\} \quad e_s \preceq e_t \quad \{v_s, v_t.\ Q\} \]

precondition \quad postcondition

source expression (unoptimized) \quad target expression (optimized)

coinductive simulation
• coinduction
• reasoning about UB
• flexible stuttering

separation logic
• compositional proof rules
• ownership reasoning
  with custom resources \textit{Iris}
Key ingredient: a powerful simulation relation

\[ \{ P \} \ e_s \succeq e_t \ \{ v_s, v_t. \ Q \} \]

- coinductive simulation
  - coinduction
  - reasoning about UB
  - flexible stuttering

- separation logic
  - compositional proof rules
  - ownership reasoning
    with custom resources
Source and target reasoning with flexible stuttering

\[ \{P\} \ e_s \geq e_t \ \{\nu_s, \nu_t. \ Q\} \]
Source and target reasoning with flexible stuttering

\[
\{P\} e_s \geq e_t \{v_s, v_t. Q\}
\]

focus source

\[
\{P\} e_s \{e'_s. R\}^{src}
\]

with rules of total separation logic

\[
\{P\} e_t \{e'_t. R\}^{tgt}
\]

focus target
Source and target reasoning with flexible stuttering

\[
\{P\} e_s \geq e_t \{v_s, v_t. Q\}
\]

focus source

\[
\{P\} e_s \{e'_s. R\}^{src}
\]

focus target

\[
\{P\} e_t \{e'_t. R\}^{tgt}
\]

with rules of total separation logic

Enabled by a flexible **implicit stuttering** mechanism without explicit step counting!
Key ingredient: a powerful simulation relation

\[ \begin{align*}
\{ P \} \ e_s &\preceq e_t \ \{ v_s, v_t. \ Q \}
\end{align*} \]

coinductive simulation
- coinduction
- reasoning about UB
- flexible stuttering

separation logic
- compositional proof rules
- ownership reasoning
  with custom resources Iris
Ownership is useful for justifying optimizations

```
let x := new(42) in
call f ();
```

```
let x := new(42) in
```

```
call f ();
```

```
\*x
```

```
42
```
Ownership is useful for justifying optimizations

\[
\{ \text{True} \} \\
\text{let } x := \text{new}(42) \text{ in} \quad \text{let } x := \text{new}(42) \text{ in} \\
\text{call } f(); \quad \geq \quad \text{call } f(); \\
\text{*}_{x} \quad \geq \quad 42 \\
\{ v_s, v_t. \ v_s = v_t = 42 \} 
\]
Ownership is useful for justifying optimizations

\[
\begin{align*}
\{ \text{True} \} \\
\text{let } x := \text{new}(42) \text{ in} & \quad \text{let } x := \text{new}(42) \text{ in} \\
\{ x \mapsto_{\text{src}} 42 \} \ast x \mapsto_{\text{tgt}} 42 & \quad {\geq} \quad \text{call } f (); \\
\ast x & \quad \text{call } f (); \\
\{ v_s, v_t. v_s = v_t = 42 \} & \\
\end{align*}
\]
Ownership is useful for justifying optimizations

\[
\begin{align*}
\{ \text{True} \} \\
\text{let } x := \text{new}(42) \text{ in } \quad \text{let } x := \text{new}(42) \text{ in }
\begin{cases}
  x \mapsto_{\text{src}} 42 \times x \mapsto_{\text{tgt}} 42 \\
\end{cases}
\text{call } f(); \\
\quad \succeq \\
\text{call } f(); \\
\begin{cases}
  x \mapsto_{\text{src}} 42 \times x \mapsto_{\text{tgt}} 42 \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\ast_x \\
\{ v_s, v_t. v_s = v_t = 42 \}
\end{align*}
\]
Ownership is useful for justifying optimizations

\[
\{\text{True}\} \\
\text{let } x := \text{new}(42) \text{ in } \text{call } f();\\
\{x \mapsto_{\text{src}} 42 * x \mapsto_{\text{tgt}} 42\} \\
\geq \text{call } f(); \\
\{x \mapsto_{\text{src}} 42 * x \mapsto_{\text{tgt}} 42\} \\
\star_{x} 42 \\
\{v_s, v_t. v_s = v_t = 42\}
\]

Unknown code must respect the ownership principles of our logic!
Key ingredient: a powerful simulation relation

\[ \{ P \} \; e_s \succeq e_t \{ v_s, v_t. \; Q \} \]

coinductive simulation
- coinduction
- reasoning about UB
- flexible stuttering

separation logic
- compositional proof rules
- ownership reasoning with custom resources
Interaction with coinduction

\{\text{True}\}

let x := new(42) in

while call f (\*x) do

() \od

\{v_s, v_t. v_s = v_t = ()\}

let x := new(42) in

while call f (42) do

() \od

\{v_s, v_t. v_s = v_t = ()\}
Interaction with coinduction

\[
\{ \text{True} \} \\
\text{let } x := \text{new}(42) \text{ in} \\
\{ x \mapsto_{\text{src}} 42 \ast x \mapsto_{\text{tgt}} 42 \} \\
\text{while call } f \left( \ast x \right) \text{ do} \quad \geq \quad \text{while call } f \left( 42 \right) \text{ do} \quad () \quad () \\
\text{od} \quad \text{od} \\
\{ v_s, v_t. v_s = v_t = () \} 
\]
Reasoning about loops: coinduction

\[ W_s = \text{while } c_s \text{ do } e_s \text{ od} \quad W_t = \text{while } c_t \text{ do } e_t \text{ od} \]

\[ \{ I \} W_s \succeq W_t \{ v_s, v_t, Q \} \]

loop invariant
Reasoning about loops: coinduction

\[ W_s = \text{while } c_s \text{ do } e_s \text{ od} \quad W_t = \text{while } c_t \text{ do } e_t \text{ od} \]

new proof goal

\[
\{I\} \quad \text{if } c_s \text{ then } e_s; W_s \text{ else } () \quad \geq \quad \text{if } c_t \text{ then } e_t; W_t \text{ else } ()
\]

\[
\{e'_s, e'_t. (\exists v_s, v_t. e'_s = v_s * e'_t = v_t * Q)\}
\]

\[
\{I\} \quad W_s \geq W_t \{v_s, v_t. Q\}
\]

loop invariant
Reasoning about loops: coinduction

\[ W_s = \text{while } c_s \text{ do } e_s \text{ od} \quad W_t = \text{while } c_t \text{ do } e_t \text{ od} \]

new proof goal

\[ \{ I \} \quad \text{if } c_s \text{ then } e_s; W_s \text{ else } () \quad \text{if } c_t \text{ then } e_t; W_t \text{ else } () \]

\[ \{ e'_s, e'_t. (\exists v_s, v_t. e'_s = v_s * e'_t = v_t * Q \quad \lor \quad (e'_s = W_s * e'_t = W_t * I)) \} \]

{\{ I \}} W_s \succeq W_t \{ v_s, v_t. Q \}

loop invariant

allows use of coinduction hypothesis
Interaction with coinduction

\[
\{ \text{True} \} \\
\text{let } x := \text{new}(42) \text{ in} \quad \text{let } x := \text{new}(42) \text{ in} \\
\{ x \mapsto_{\text{src}} 42 \ast x \mapsto_{\text{tgt}} 42 \} \\
\text{while call } f (x) \text{ do} \quad \geq \quad \text{while call } f (42) \text{ do} \\
\quad () \\
\quad \text{od} \\
\quad \text{od} \\
\{ \nu_s, \nu_t. \nu_s = \nu_t = () \}
\]

Pick invariant \( I \triangleq x \mapsto_{\text{src}} 42 \ast x \mapsto_{\text{tgt}} 42 \)
Interaction with coinduction

Ownership reasoning is a powerful tool in combination with coinductive simulations!

Pick invariant \( I \triangleq x \mapsto_{\text{src}} 42 \ast x \mapsto_{\text{tgt}} 42 \)
Key ingredient: a powerful simulation relation

\{ P \} \ e_s \cong \ e_t \ \{ v_s, v_t. \ Q \} 

coinductive simulation
- coinduction
- reasoning about UB
- flexible stuttering

separation logic
- compositional proof rules
- ownership reasoning
- with custom resources
Simuliris: a separation logic-based simulation framework

- Soundness: fair termination-preserving contextual refinement
- Proof rules for verifying optimizations: coinduction, . . .

Fully mechanized in the Coq proof assistant based on the Iris framework

Stacked Borrows for Rust [Jung et al., 2020] + concurrency
Optimizations with external locations: motivating example

```
fn foo(x) {
    x ← 41;
    x ← 42;
    *x
}
```

```
fn foo_opt(x) {
    ≥
    x ← 42;
    42
}
```
Optimizations with external locations: motivating example

**in separation logic verification:** assume ownership in precondition

```plaintext
fn foo(x) {
  x ← 41;
  x ← 42;
  *x
}

fn foo_opt(x) {
  x ← 42;
  42
}
```
Optimizations with external locations: motivating example

in separation logic verification: assume ownership in precondition

```python
fn foo(x) {
    x ← 41;
    x ← 42;
    *x
}
```

```python
fn foo_opt(x) {
    x ← 42;
    42
}
```

in compiler optimizations: surrounding code is **not** cooperative!

```python
fn foo(x) {
    x ← 41;
    x ← 42;
    *x
}
```

```python
fn foo_opt(x) {
    x ← 42;
    42
}
```
Optimizations with external locations: motivating example

in separation logic verification: assume ownership in precondition

```plaintext
fn foo(x) {
    \( x \leftrightarrow_{\text{src}} z \land x \leftrightarrow_{\text{tst}} z \)
    x ← 41;
    x ← 42;
    *x
}
```

```plaintext
fn foo_opt(x) {
    x ← 42;
    42
}
```

in compiler optimizations: surrounding code is not cooperative!

```plaintext
fn foo(x) {
    \( x_s \approx x_t \)
    x ← 41;
    x ← 42;
    *x
}
```

```plaintext
fn foo_opt(x) {
    x ← 42;
    42
}
```
interaction protocol with unknown code: public value relation

**contract**: similar values $v_s \approx v_t$ in source and target

For integers: $z_s \approx z_t \triangleq z_s = z_t$

For memory locations $\ell_s \approx \ell_t$:

- **contract**: stored values are related by $\approx$
- **accessible by anyone** as long as the contract is observed
Interaction protocol with unknown code: public value relation

**contract:** similar values $v_s \approx v_t$ in source and target

for integers: $z_s \approx z_t \overset{\triangle}{=} z_s = z_t$

for memory locations $\ell_s \approx \ell_t$:

- **contract:** stored values are related by $\approx$
- **accessible by anyone** as long as the contract is observed

How can we use this to justify optimizations?
Interaction protocol with unknown code: public value relation

**Contract:** similar values \( v_s \approx v_t \) in source and target

**Idea:** we can break the contract as long as no other thread will notice

- **Contract:** stored values are related by \( \approx \)
- **Accessible by anyone** as long as the contract is observed

How can we use this to justify optimizations?
Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of $\ell_s$ and $\ell_t$. 
Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of $\ell_s$ and $\ell_t$.

An unsynchronized write $\ell_s \leftarrow _{-}$ is reachable:

$\Rightarrow$ all concurrent accesses would be conflicting

$\Rightarrow$ obtain exclusive ownership $\ell_s \mapsto_{\text{src}} v_s, \ell_t \mapsto_{\text{tgt}} v_t$
Acquiring ownership on writes, formally: first attempt

\[ \{ \ell_s \approx \ell_t \ast P \} K[ \ell_s \leftarrow v_0 ] \succeq e_t \{ \Phi \} \]

- **public locations**
- **unsynchronized write in the source**
Acquiring ownership on writes, formally: first attempt

\[ \forall v_s, v_t. \{ l_s \xrightarrow{\text{src}} v_s * l_t \xrightarrow{\text{tgt}} v_t * v_s \approx v_t * P \} K[l_s \leftarrow v_0] \succeq e_t \{ \Phi \} \]

\[ \{ l_s \approx l_t * P \} K[l_s \leftarrow v_0] \succeq e_t \{ \Phi \} \]

 obtai n ownership

 public locations

 unsynchronized write in the source
Proving the optimization with ownership

\{ x_s \approx x_t \}

\begin{align*}
    x_s &\leftarrow 41; \\
    x_s &\leftarrow 42; \quad \geq \quad x_t \leftarrow 42; \\
    *x_s &\leftarrow 42
\end{align*}
Proving the optimization with ownership

\[
\begin{align*}
\{ x_s \approx x_t \} \\
\{ x_s \leftarrow _{\text{src}}^{} z \star x_t \leftarrow _{\text{tgt}}^{} z \}
\end{align*}
\]

\[
x_s \leftarrow 41;
\]

\[
x_s \leftarrow 42; \quad \geq \quad x_t \leftarrow 42;
\]

\[
* x_s \quad 42
\]
Proving the optimization with ownership

\[
\begin{align*}
\{x_s \approx x_t\} \\
\{x_s \mapsto_{\text{src}} z * x_t \mapsto_{\text{tgt}} z\}
\end{align*}
\]

\[
x_s \leftarrow 41;
\]

contract temporarily broken

\[
x_s \leftarrow 42;\quad x_t \leftarrow 42;
\]

\[
* x_s
\]

\[
v_s, v_t = 42 \quad \geq \quad x_t \leftarrow 42;
\]

What prevents us from acquiring ownership multiple times?
Proving the optimization with ownership

\[
\{ x_s \approx x_t \} \\
\{ x_s \mapsto_{\text{src}} z \ast x_t \mapsto_{\text{tgt}} z \}
\]

\[
x_s \leftarrow 41; \\
\{ x_s \mapsto_{\text{src}} 41 \ast x_t \mapsto_{\text{tgt}} z \}
\]

\[
x_s \leftarrow 42; \\
\{ x_s \mapsto_{\text{src}} 42 \ast x_t \mapsto_{\text{tgt}} 42 \}
\]

\[
* x_s \\
\geq \\
x_t \leftarrow 42; \\
42
\]
Proving the optimization with ownership

\[
\{ x_s \approx x_t \} \\
\{ x_s \overset{\text{src}}{\mapsto} z \ast x_t \overset{\text{tgt}}{\mapsto} z \}
\]

\[
x_s \leftarrow 41; \\
\{ x_s \overset{\text{src}}{\mapsto} 41 \ast x_t \overset{\text{tgt}}{\mapsto} z \}
\]

\[
x_s \leftarrow 42; \\
\{ x_s \overset{\text{src}}{\mapsto} 42 \ast x_t \overset{\text{tgt}}{\mapsto} 42 \}
\]

\[
\{ v_s, v_t. v_s = v_t = 42 \ast x_s \approx x_t \}
\]

What prevents us from acquiring ownership multiple times?
Proving the optimization with ownership

\[
\begin{align*}
\{x_s \approx x_t\} \\
\{x_s \mapsto^\text{src} z * x_t \mapsto^\text{tgt} z\} \\
\{x_s \mapsto^\text{src} z' * x_t \mapsto^\text{tgt} z' * x_s \mapsto^\text{src} z * x_t \mapsto^\text{tgt} z\} \\
x_s \leftarrow 41; \\
\{x_s \mapsto^\text{src} 41 * x_t \mapsto^\text{tgt} z\} \\
x_s \leftarrow 42; \\
\{x_s \mapsto^\text{src} 42 * x_t \mapsto^\text{tgt} 42\} \\
* x_s \\
\{v_s, v_t. v_s = v_t = 42 * x_s \approx x_t\}
\end{align*}
\]

What prevents us from acquiring ownership multiple times?
Proving the optimization with ownership

\[
\{x_s \approx x_t\}
\]

\[
\{x_s \mapsto^{\text{src}} Z \ast x_t \mapsto^{\text{tgt}} Z\}
\]

\[
\{x_s \mapsto^{\text{src}} Z' \ast x_t \mapsto^{\text{tgt}} Z' \ast x_s \mapsto^{\text{src}} Z \ast x_t \mapsto^{\text{tgt}} Z\}
\]

\[x_s \leftarrow 41;\]

\[
\{x_s \mapsto^{\text{src}} 41 \ast x_t \mapsto^{\text{tgt}} Z\}
\]

The acquisition rule is unsound! 😞

\[
\{x_s \mapsto^{\text{src}} 42 \ast x_t \mapsto^{\text{tgt}} 42\}
\]

\[\ast x_s\]

\[
\{v_s, v_t. v_s = v_t = 42 \ast x_s \approx x_t\}
\]

What prevents us from acquiring ownership multiple times?
Solution: avoiding duplication of ownership

Track locations exploited by the current thread $\pi$: exploit$_{\pi} C$

\[ \forall v_t, v_s . \{ l_s \rightarrow^{\text{src}} v_s * l_t \rightarrow^{\text{tgt}} v_t * v_s \approx v_t * \} \]

\[ \{ l_s \approx l_t * \} \]

\[ P \} K[ l_s \leftarrow v_0 ] \succeq_{\pi} e_t \{ \Phi \} \]

obtain ownership

unsynchronized write in the source

public locations
Solution: avoiding duplication of ownership

Track locations exploited by the current thread $\pi$: $\text{exploit}_\pi C$

\[
\forall v_t, v_s. \{ l_s \xrightarrow{\text{src}} v_s * l_t \xrightarrow{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi (C, l_s \xrightarrow{\text{W}}) * P \} \ K[ l_s \leftarrow v_0 ] \succeq_\pi e_t \{ \Phi \}
\]

- obtain ownership
- exploit once
- remember $l_s$
- public locations
- track exploited locations for current thread
- unsynchronized write in the source
Solution: avoiding duplication of ownership

Track locations exploited by the current thread $\pi$: $\text{exploit}_\pi C$

- Defined with custom Iris ghost state

- Obtain ownership

- Exploit once

- Remember $\ell_s$

- Public locations

- Track exploited locations for current thread

- Unsynchronized write in the source

\[ \forall v_t, v_s. \{ \ell_s \mapsto v_s \} \ast v_t \mapsto v_s \approx v_t \ast \text{exploit}_\pi (C, \ell_s \mapsto W) \ast P \} \ K[\ell_s \leftarrow v_0] \succeq_{\pi} e_t \{ \Phi \} \]

\[ \{ \ell_s \approx \ell_t \ast \text{exploit}_\pi C \ast P \} \ K[\ell_s \leftarrow v_0] \succeq_{\pi} e_t \{ \Phi \} \]
Solution: avoiding duplication of ownership

Track locations exploited by the current thread $\pi$: $\text{exploit}_\pi C$

defined with custom
his ghost state

generalization: unsynchronized access just needs to be reachable (not the directly next instruction)

obtain ownership

$\forall v_{ts}, v_s. \{l_s \mapsto_{\text{src}} v_s * l_t \mapsto_{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi (C, l_s \mapsto W) * P \} K[l_s \leftarrow v_0] \succeq_\pi e_t \{\Phi\}$

unsynchronized write
in the source

public locations

track exploited locations for current thread

unsynchronized write
in the source

l_s \notin C

remember l_s

{l_s \approx l_t * \text{exploit}_\pi C * P \} K[l_s \leftarrow v_0] \succeq_\pi e_t \{\Phi\}$
Solution: avoiding duplication of ownership

Track locations exploited by the current thread $\pi$: exploit $\pi$ $C$

defined with custom Iris ghost state

generalization: unsynchronized access just needs to be reachable (not the directly next instruction)

rule for reads: obtain fractional (read-only) ownership

unsynchronized write in the source

obtain ownership

public locations

track exploited locations for current thread

unsynchronized write in the source

unsynchronized write in the source

unsynchronized write in the source
Maintaining & releasing ownership on synchronization

We can maintain ownership until the thread observably synchronizes.

action (potentially) visible by other threads
We can maintain ownership until the thread observably synchronizes.

Example: rule for atomic writes

\[
\{ \ell_s \approx \ell_t \ast v_s \approx v_t \ast \text{exploit}_\pi \emptyset \} \\
\ell_s \leftarrow^{\text{sc}} v_s \succsim_\pi \ell_t \leftarrow^{\text{sc}} v_t \\
\{ v'_s, v'_t \cdot \text{exploit}_\pi \emptyset \}
\]
Verifying the motivating example

```c
int i = 0; int sum = *y;
while (i != *x) {
    i += 1; sum += *y;
}
return sum;
```

```c
int n = *x;
int m = *y;
int i = 0; int sum = m;
while (i != n) {
    i += 1; sum += m;
}
return sum;
```
Verifying the motivating example

\{x_s \approx x_t \land y_s \approx y_t\}

let \( (m, n) := (\ast y, \ast x) \) in

let \( (i, \text{sum}) := (\text{new}(0), \text{new}(\ast y)) \) in

while \( \ast i \neq \ast x \) do

\( i \leftarrow \ast i + 1; \)

\( \text{sum} \leftarrow \ast \text{sum} + \ast y \)

od; \( \ast \text{sum} \)

let \( (m, n) := (\text{new}(0), \text{new}(m)) \) in

let \( (i, \text{sum}) := (\text{new}(0), \text{new}(m)) \) in

while \( \ast i \neq n \) do

\( i \leftarrow \ast i + 1; \)

\( \text{sum} \leftarrow \ast \text{sum} + m \)

od; \( \ast \text{sum} \)

\{v_s, v_t. v_s \approx v_t\}

1. Obtain ownership of \( x \) and \( y \) due to unsynchronized reads in the source

2. Initiate coinduction
Simuliris: a separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...
Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement
Soundness: fair termination-preserving contextual refinement

\[ \text{fair termination-preserving contextual refinement} \]

- optimized expression
  - can terminate with result
- unoptimized expression
  - can terminate with similar result

Core soundness proof: proved once and for all!
Soundness: fair termination-preserving contextual refinement

**fair termination-preserving contextual refinement**

<table>
<thead>
<tr>
<th>optimized expression</th>
<th>unoptimized expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>can terminate with result</td>
<td>can terminate with similar result</td>
</tr>
<tr>
<td>can diverge <em>under fair scheduling</em></td>
<td>can diverge <em>under fair scheduling</em></td>
</tr>
</tbody>
</table>
Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

for any surrounding program, assuming no UB in unoptimized program

<table>
<thead>
<tr>
<th>optimized expression</th>
<th>unoptimized expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>can terminate with result</td>
<td>can terminate with similar result</td>
</tr>
<tr>
<td>can diverge under fair scheduling</td>
<td>can diverge under fair scheduling</td>
</tr>
</tbody>
</table>
Soundness: fair termination-preserving contextual refinement

fair termination-preserving contextual refinement

for any surrounding program, assuming no UB in unoptimized program

<table>
<thead>
<tr>
<th>optimized expression</th>
<th>unoptimized expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>can terminate with result</td>
<td>can terminate with similar result</td>
</tr>
<tr>
<td>can diverge under fair scheduling</td>
<td>can diverge under fair scheduling</td>
</tr>
</tbody>
</table>

Core soundness proof: proved once and for all!
Simuliris: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, . . .

Stacked Borrows for Rust [Jung et al., 2020] + concurrency

logic for data race based optimizations

fully mechanized in the Coq proof assistant based on the Iris framework
Simuliris: a separation logic-based simulation framework

- logic for data race based optimizations
- Stacked Borrows for Rust [Jung et al., 2020] + concurrency

**Simuliris**: separation logic-based simulation framework

- soundness: fair termination-preserving contextual refinement
- proof rules for verifying optimizations: coinduction, ...

Fully mechanized in the Coq proof assistant based on the Iris framework

https://gitlab.mpi-sws.org/iris/simuliris

Thanks for listening!
Generalization: reachability of an unsynchronized write

The argument even works if the write is just (unconditionally) reachable!

\[ \forall v_t, v_s. \{ \ell_s \mapsto_{src} v_s \ast \ell_t \mapsto_{tgt} v_t \ast v_s \approx v_t \ast \text{exploit}_\pi (C, \ell_s \mapsto W) \ast P \} \ e_s \succeq_\pi e_t \{ \Phi \} \]

\[ \{ \ell_s \approx \ell_t \ast \text{exploit}_\pi C \ast P \} \ e_s \succeq_\pi e_t \{ \Phi \} \]

- Generalization: reachability of an unsynchronized write
- The argument even works if the write is just (unconditionally) reachable!
- Generalization: reachability of an unsynchronized write
- The argument even works if the write is just (unconditionally) reachable!
Generalization: reachability of an unsynchronized write

The argument even works if the write is just (unconditionally) reachable!

\[ \forall v_t, v_s. \{ l_s \mapsto^{\text{src}} v_s * l_t \mapsto^{\text{tgt}} v_t * v_s \approx v_t * \text{exploit}_\pi (C, l_s \mapsto W) * P \} \quad e_s \succeq_{\pi} e_t \{ \Phi \} \]

A similar rule holds for reads!
Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $l_s \approx l_t$, we temporarily obtain ownership of $l_s$ and $l_t$.

An unsynchronized read $*l_s$ is reachable:
$\Rightarrow$ concurrent write accesses would be conflicting
$\Rightarrow$ obtain **fractional** ownership $l_s \mapsto_{src} q v_s$, $l_t \mapsto_{tgt} q v_t$
Ownership acquisition on unsynchronized accesses

When the source program does an unsynchronized access to $\ell_s \approx \ell_t$, we temporarily obtain ownership of $\ell_s$ and $\ell_t$.

<table>
<thead>
<tr>
<th>Do we ever have to give up ownership again?</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ concurrent write accesses would be conflicting</td>
</tr>
<tr>
<td>⇒ obtain <strong>fractional</strong> ownership $\ell_s \mapsto_{q}^\text{src} \nu_s$, $\ell_t \mapsto_{q}^\text{tgt} \nu_t$</td>
</tr>
</tbody>
</table>
Rule for unsynchronized reads

\( \forall v_t, v_s, q. \left\{ \ell_s \not\in C \right\} e_s \rightarrow K[\ast \ell_s] e_s \rightarrow^* v_t \leftarrow q v_s \rightarrow^* v_t \leftarrow q v_s \approx v_t \ast \text{exploit}_\pi (C, \ell_s \leftarrow R(q)) \ast P \right\} e_s \subseteq e_t \{ \Phi \} \left\{ l_s \approx l_t \ast \text{exploit}_\pi C \ast P \right\} e_s \subseteq e_t \{ \Phi \} \)

obtain ownership

reach unsynchronized read in the source

exploit once

public locations

track exploited locations for current thread
Releasing ownership

\[
C(\ell_s) = W \quad \{\text{exploit}_\pi (C \setminus \ell_s) \ast P\} \quad e_s \succeq \pi \quad e_t \quad \{\Phi\}
\]

\[
\frac{\ell_s \mapsto_{\text{src}} v_s \ast \ell_t \mapsto_{\text{tgt}} v_t \ast v_s \approx v_t \ast \ell_s \approx \ell_t \ast \text{exploit}_\pi C \ast P}{e_s \succeq \pi \quad e_t \quad \{\Phi\}}
\]
Fair termination-preserving refinement

The compiler should not be allowed to perform the following transformation:

\[
\text{while } !\text{lock}(l) \; \text{do} \\
\quad () \\
\text{od; unlock}(l) \\
\text{if lock}(l) \quad \text{then unlock}(l) \quad \text{while true do} \\
\quad () \quad \text{od} \\
\text{else ()} \\
\text{if lock}(l) \quad \text{then unlock}(l) \\
\quad () \quad \text{else ()}
\]

The source program only has diverging executions under an unfair scheduler, while the target program diverges even under a fair scheduler.
In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop

```plaintext
let (i, sum) := (new(0), new(*ys)) in
while *i ≠ *xs do
    i ← *i + 1; sum ← *sum + *ys
od; *sum
```

```plaintext
let (n, m) := (*xt, *yt) in
```

```plaintext
let (i, sum) := (new(0), new(m)) in
while *i ≠ n do
    i ← *i + 1; sum ← *sum + m
od; *sum
```

Requires reasoning about potentially infinite loops!
In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop
- optimization eliminating reads and writes over unknown read-only code

\[
x_s \leftarrow 42; \\
e^{RO}; \\
* x_s \preceq_{\pi} \\
x_t \leftarrow 42; \\
e^{RO}; \\
42
\]
In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop
- optimization eliminating reads and writes over unknown read-only code

\[ x_s \leftarrow 42; \quad x_t \leftarrow 42; \]
\[ \ast^{sc} y_s; \quad \ast^{sc} y_t; \]
\[ \ast x_s \quad \ast x_t \]
\[ \preceq_{\pi} \quad 42 \]

Not supported by CAS/Concurrent CompCert!
In the paper: proofs of optimizations relying on data races

- optimization hoisting reads out of a while loop
- optimization eliminating reads and writes over unknown read-only code
- eliminations and reorderings using data races by [Ševčík, 2009]
In the paper: new optimization proofs for Stacked Borrows

Stacked Borrows: an experimental aliasing model for Rust
- determines which kinds of memory accesses are allowed
- enables powerful optimizations
In the paper: new optimization proofs for Stacked Borrows

**Stacked Borrows**: an experimental aliasing model for Rust
- determines which kinds of memory accesses are allowed
- enables powerful optimizations

Using Simuliris, we have...
- extended the optimization proofs by [Jung et al., 2020] to concurrent environments
- developed a new proof of an optimization involving loops:

```rust
// x: &i32, g: &Fn() -> (),
// f: &Fn(i32) -> bool
let r = *x;
while f(r) {
    g();
}
```

```rust`
while f(*x) {
    g();
}
```