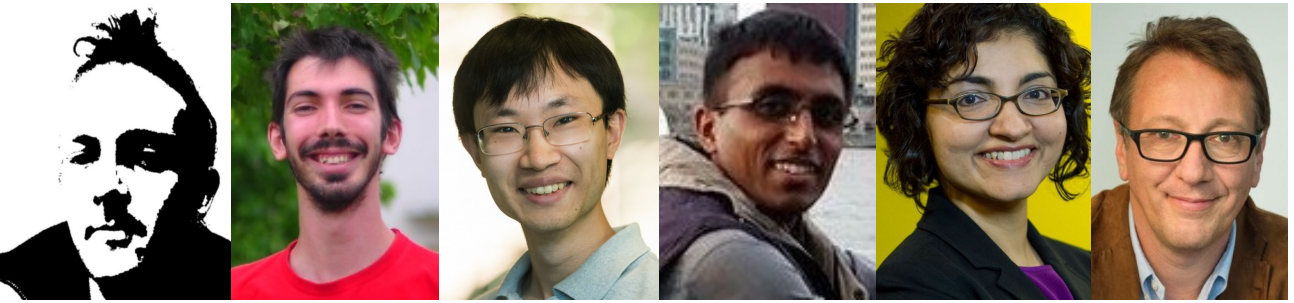
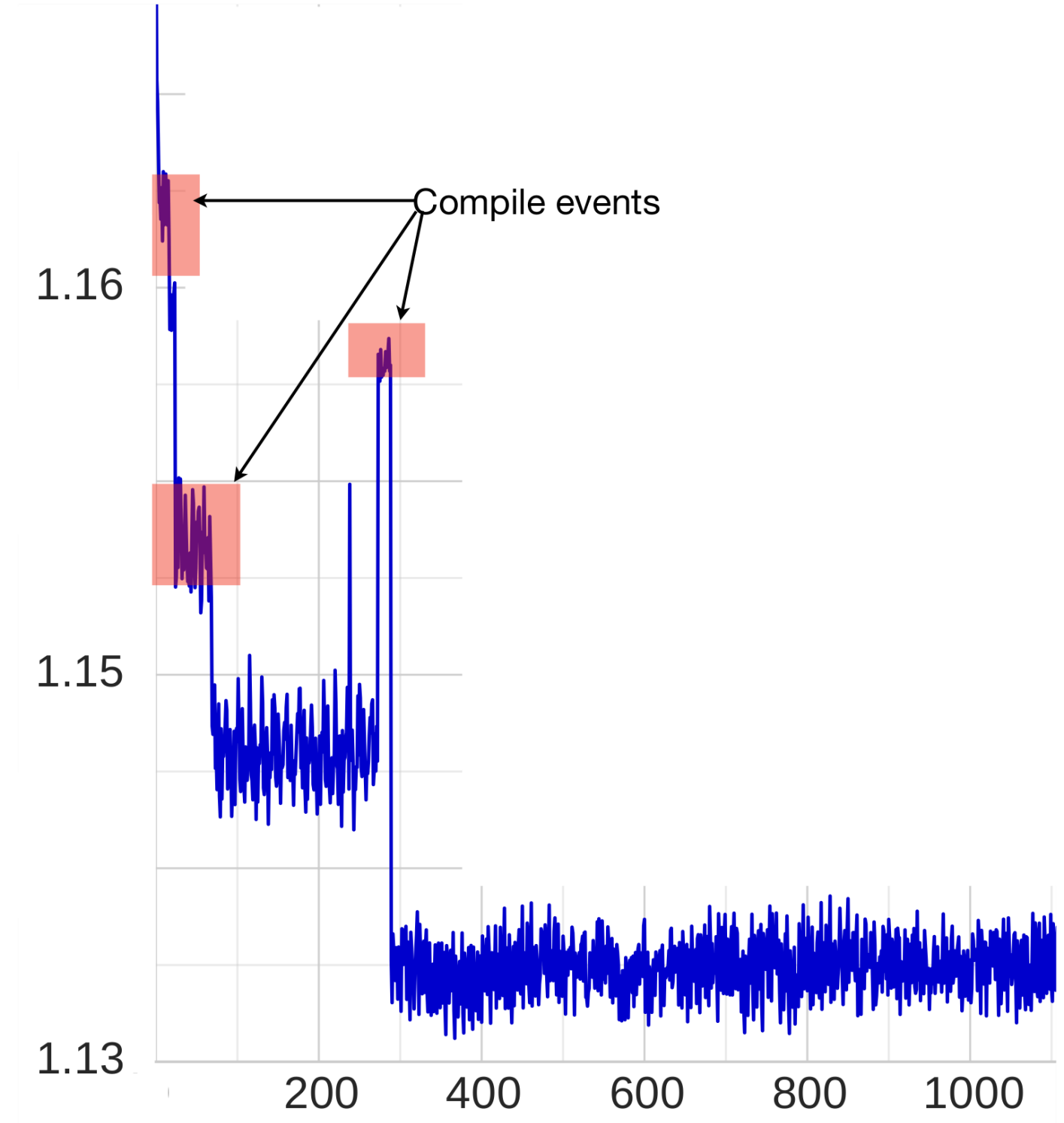
Correctness of Speculative Optimizations with Dynamic Deoptimization



Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee,
Aviral Goel, Amal Ahmed, Jan Vitek

Northeastern University,
INRIA, and CVUT



Fasta JS benchmark,
V8, Linux, i7-4790

[Barrett ea. *Virtual Machine Warmup Blows Hot and Cold*. OOPSLA17]

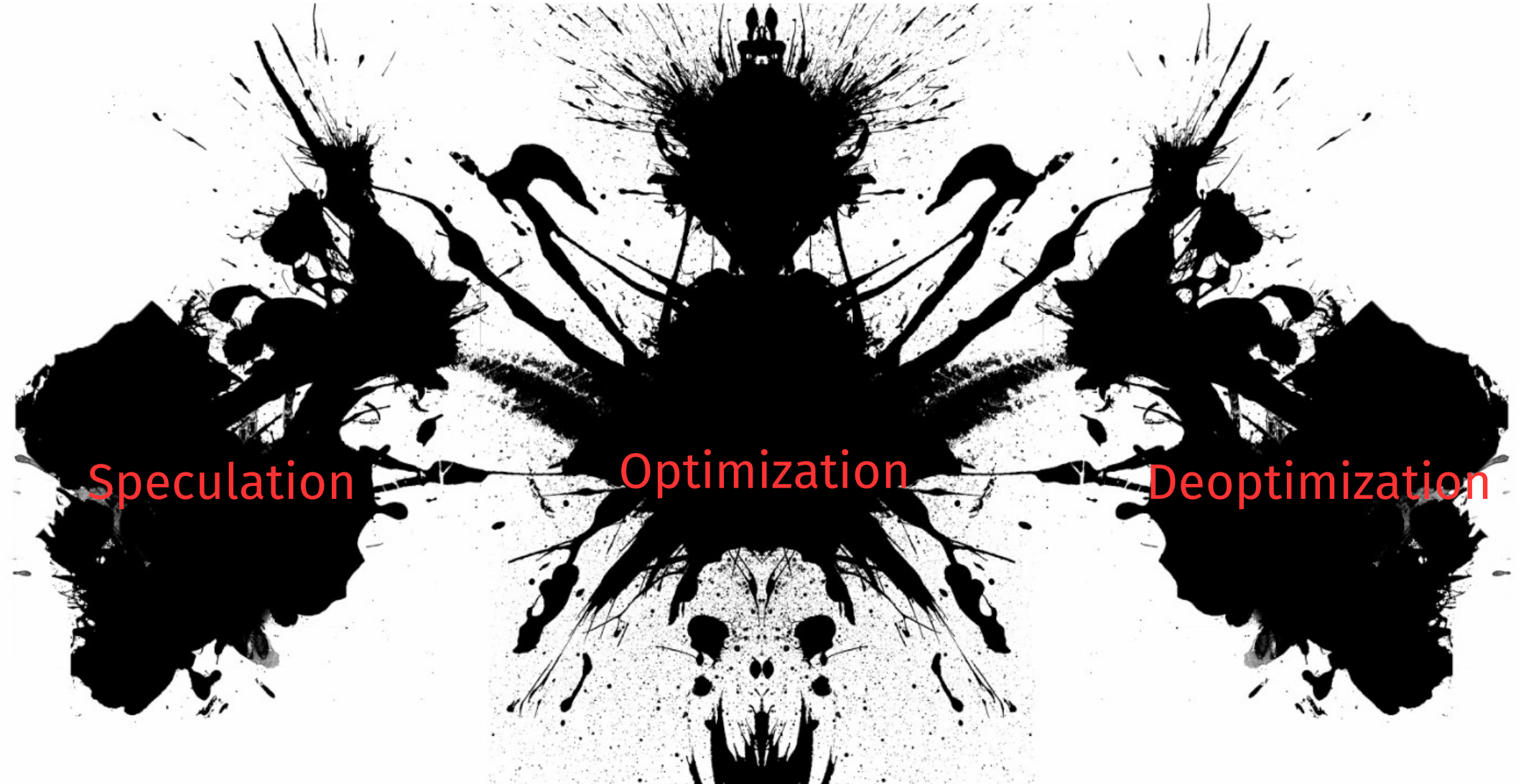
```
function sorted(x) {
  for (var i = 1; i < x.length; i++)
    if (x[i] < x[i-1])
      return false;
  return true;
}
```

```
function sorted(x) {
  for (var i = 1; i < x.length; i++)
    if (x[i] < x[i-1])
      return false;
  return true;
}

function `[]`(x,i) {
  if (typeof(x) != array) error()
    if (typeof(i) != int)
      i = convert(i, int)
  return get(x, i)
}

function `<`(a,b) {
  if (typeof(a) == float) {
    if (typeof(b) != float)
      b = convert(b, float)
    return ltf(a.val, b.val)
  }
  if (typeof(b) == int) {
    …
  …
}
```

```
function sorted(x) {
  for (var i = 1; i < x.length; i++)
    if (x[i] < x[i-1])
      return false;
  return true;
}

function `[]`(x,i) {
  if (typeof(x) != array) error()
    if (typeof(i) != int)
      i = convert(i, int)
  return get(x, i)
}

function `<`(a,b) {
  if (typeof(a) == float) {
    if (typeof(b) != float)
      b = convert(b, float)
    return ltf(a.val, b.val)
  }
  if (typeof(b) == int) {
    …
  …
}
```

```
function sorted(x) {
  for (var i = 1; i < x.length; i++)


    t1 = get(x, i)

    t2 = get(x, i-1)
    t3 = t1.val
    t4 = t3.val
    t5 = ltf(t3, t4)
    if (t5) return 0

  return 1
}
```

```
function sorted(x) {
  for (var i = 1; i < x.length; i++)

    DeoptIf(typeof x != floatarray)
    DeoptIf(OutOfBounds x, i)
    t1 = get(x, i)
    DeoptIf(OutOfBounds x, i-1)
    t2 = get(x, i-1)
    t3 = t1.val
    t4 = t3.val
    t5 = ltf(t3, t4)
    if (t5) return 0

  return 1
}
```


Speculation
Optimization
Deoptimization

When are Speculative Optimizations with Dynamic Deoptimization correct?

Sourir

Sourir

$$
\begin{aligned}
i \ ::= \\
&\mid \ \textbf{var}\ x = e \\
&\mid \ \textbf{drop}\ x \\
&\mid \ x \leftarrow e \\
&\mid \ \textbf{array}\ x[e] \\
&\mid \ \textbf{array}\ x = [e^*] \\
&\mid \ x[e_1] \leftarrow e_2 \\
&\mid \ \textbf{branch}\ e\ L_1\ L_2 \\
&\mid \ \textbf{goto}\ L \\
&\mid \ \textbf{print}\ e \\
&\mid \ \textbf{read}\ x \\
&\mid \ \textbf{call}\ x = e(e^*) \\
&\mid \ \textbf{return}\ e
\end{aligned}
$$

https://github.com/reactorlabs/sourir

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2

         var off = 2
         return x[off+i]

 V3

         return x[2]
```

```
get(x, i)
```

V1

```
     var off = 2
L0   branch (x = nil) L2 L1
L1   return x[off+i]
L2   return nil
```

V2

```
var off = 2
return x[off+i]
```

V3

```
return x[2]
```

```
get(x, i)

V1      var off = 2
    L0  branch (x = nil) L2 L1
    L1  return x[off+i]
    L2  return nil

V2

        var off = 2
        return x[off+i]

V3

        return x[2]
```

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2

         var off = 2
         return x[off+i]

 V3

         return x[2]
```

Assume

```
assume
    e, …                          guards
  else
    F.V.L [x=e, …]                frame synthesis
   (F.V.L x [x=e, …])*
```

Assume

```
assume
    e, …                    guards
  else
    F.V.L [x=e, …]          frame synthesis
```

Sourir

Assume

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2      var off = 2
    L0   assume (x ≠ nil)
           else get.V1.L0 [x=x,i=i,off=off]
         return x[off+i]

 V3 L0   assume (x ≠ nil, i = 0)
           else get.V2.L0 [x=x,i=i,off=2]
         return x[2]
```

Sourir

Assume

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2      var off = 2
    L0   assume (x ≠ nil)
           else get.V1.L0 [x=x,i=i,off=off]
         return x[off+i]

 V3 L0   assume (x ≠ nil, i = 0)
           else get.V2.L0 [x=x,i=i,off=2]
         return x[2]
```

Sourir

Assume

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2      var off = 2
    L0   assume (x ≠ nil)
           else get.V1.L0 [x=x,i=i,off=off]
         return x[off+i]

 V3 L0   assume (x ≠ nil, i = 0)
           else get.V2.L0 [x=x,i=i,off=2]
         return x[2]
```

Sourir

Assume

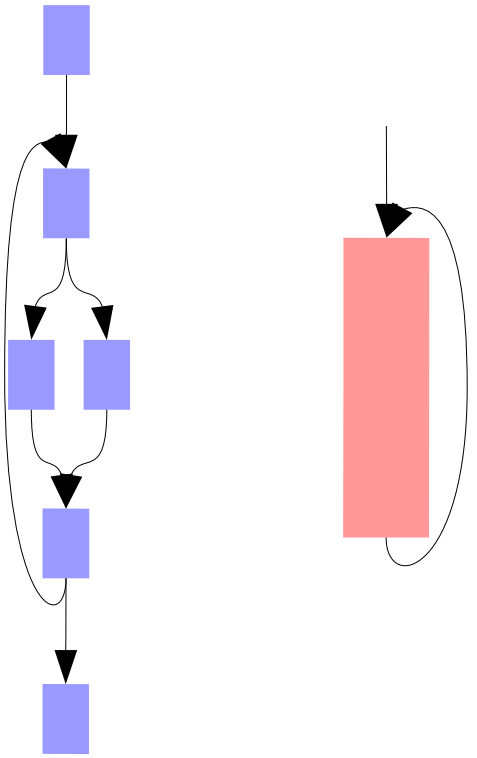
```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2      var off = 2
    L0   assume (x ≠ nil)
           else get.V1.L0 [x=x,i=i,off=off]
         return x[off+i]

 V3 L0   assume (x ≠ nil, i = 0)
           else get.V2.L0 [x=x,i=i,off=2]
         return x[2]
```

Sourir

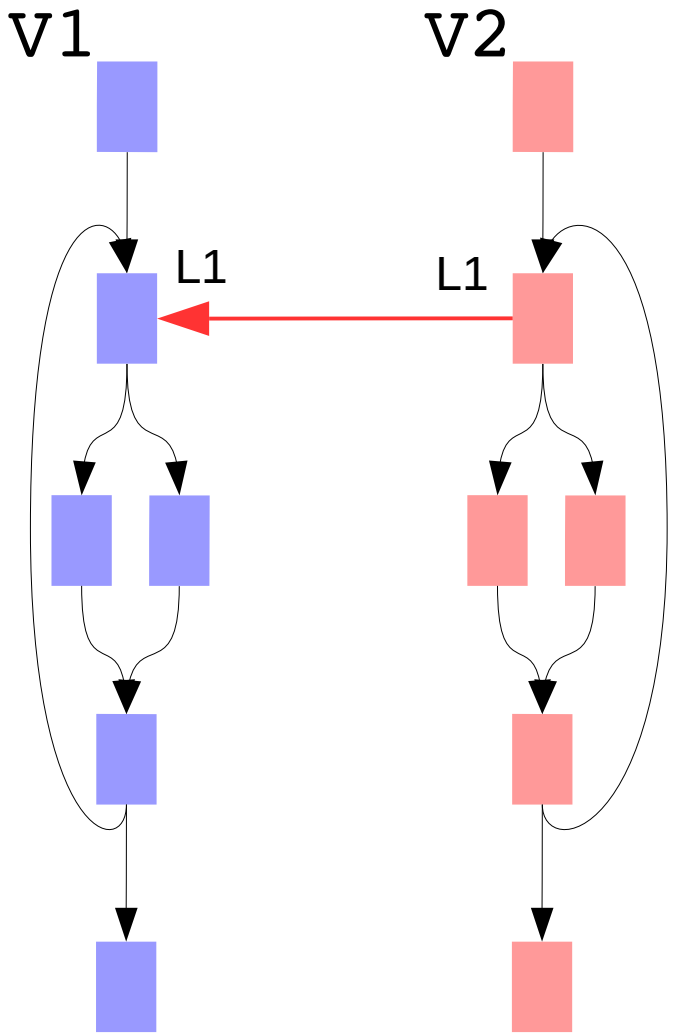
Assume

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2      var off = 2
    L0   assume (x ≠ nil)
           else get.V1.L0 [x=x,i=i,off=off]
         return x[off+i]

 V3 L0   assume (x ≠ nil, i = 0)
           else get.V2.L0 [x=x,i=i,off=2]
         return x[2]
```

Writing a JIT

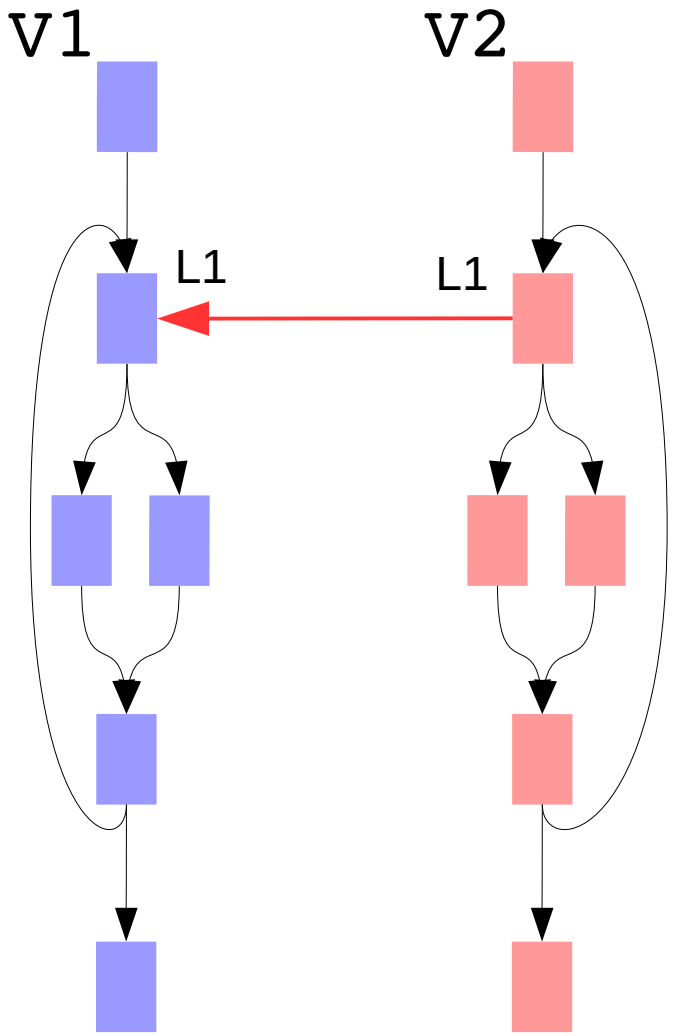


Overview



1. Create identical copy

Overview

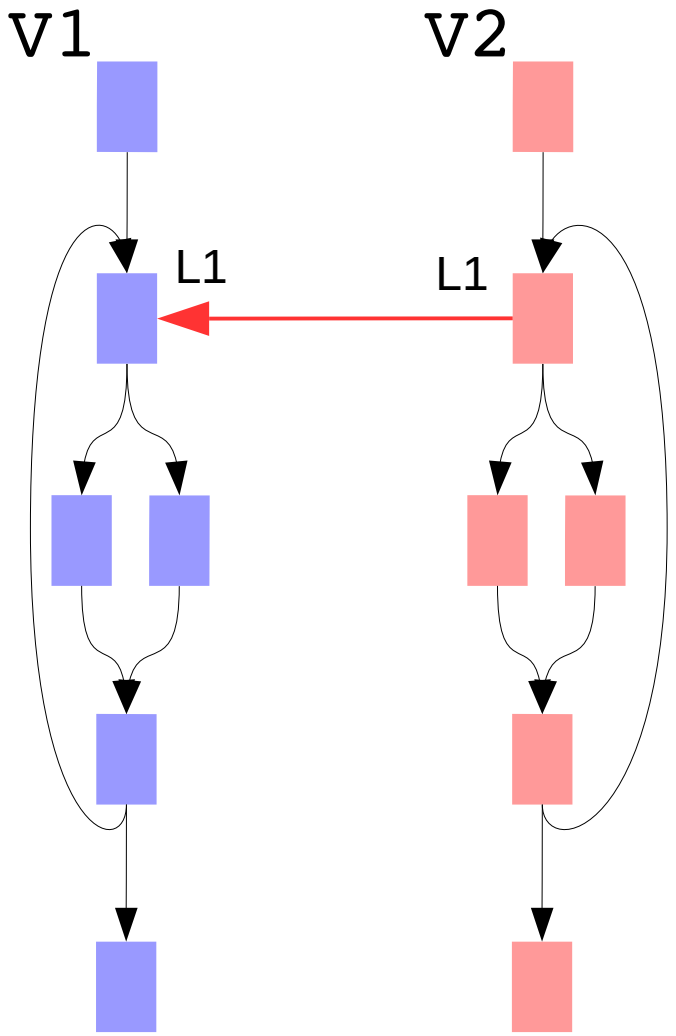


1. Create identical copy

2. Add empty assumes

```
assume true
    else F.V1.L1 [x=x]
```

Overview



1. Create identical copy

2. Add empty assumes

```
assume true
    else F.V1.L1 [x=x]
```

3. Optimize code

   by adding guards

Writing a JIT

Copy

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil
```

Writing a JIT

Copy

```
get(x, i)

 V1       var off = 2
     L0   branch (x = nil) L2 L1
     L1   return x[off+i]
     L2   return nil

 V2       var off = 2


     L0   branch (x = nil) L2 L1
     L1   return x[off+i]
     L2   return nil
```

Adding Empty Assume

```
get(x, i)

 V1      var off = 2
    L0   branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil

 V2      var off = 2
    L0   assume true
           else get.V1.L0 [x=x,i=i,off=off]
         branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil
```

Adding Empty Assume

```
get(x, i)

 V1      …

 V2      var off = 2
    L0   assume true
           else get.V1.L0 [x=x,i=i,off=off]
         branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil
```

Constant Propagation

```
get(x, i)

 V1      …

 V2      var off = 2
    L0   assume true
           else get.V1.L0 [x=x,i=i,off=off]
         branch (x = nil) L2 L1
    L1   return x[off+i]
    L2   return nil
```

Constant Propagation

```
get(x, i)

 V1      …

 V2      var off = 2
    L0   assume true
           else get.V1.L0 [x=x,i=i,off= 2 ]
         branch (x = nil) L2 L1
    L1   return x[ 2 +i]
    L2   return nil
```

Speculation

```
get(x, i)

 V1      …

 V2  L0   assume true
            else get.V1.L0 [x=x,i=i,off=2]
          branch (x = nil) L2 L1
     L1   return x[2+i]
     L2   return nil
```

Speculation

```
get(x, i)

 V1      …

 V2  L0   assume true
            else get.V1.L0 [x=x,i=i,off=2]
          branch (x = nil) L2 L1
     L1   return x[2+i]
     L2   return nil
```

Speculation

```
get(x, i)

 V1      …

 V2  L0   assume true
            else get.V1.L0 [x=x,i=i,off=2]
          branch (x = nil) L2 L1
     L1   return x[2+i]
     L2   return nil
```

Speculation

```
get(x, i)

 V1      …

 V2  L0   assume (x ≠ nil)
            else get.V1.L0 [x=x,i=i,off=2]
          branch (x = nil) L2 L1
     L1   return x[2+i]
     L2   return nil
```

Speculation

```
get(x, i)

 V1      …

 V2  L0   assume (x ≠ nil)
            else get.V1.L0 [x=x,i=i,off=2]
          branch ~~(x = nil)~~ false L2 L1
     L1   return x[2+i]
     L2   return nil
```

Speculation

```
get(x, i)

 V1      …

 V2  L0   assume (x ≠ nil)
            else get.V1.L0 [x=x,i=i,off=2]
          branch (x = nil) false L2 L1
     L1   return x[2+i]
     L2   return nil
```

Inlining

```
      array vec = [0,0,1,2]
      call  res = get(vec, 1)
L0    print res
```

Inlining

```
      array vec = [0,0,1,2]
      call  res = get(vec, 1)
L0    print res
```

Inlining

```
      array vec = [0,0,1,2]
      call  res = get(vec, 1)
L0    print res
```

```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]

var res = x[2+1]
print res
```

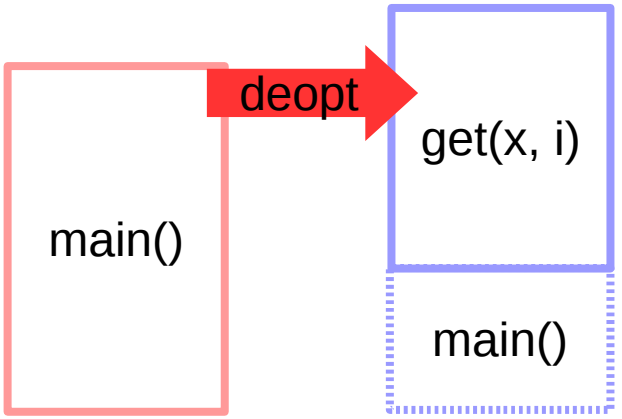
Inlining

```
      array vec = [0,0,1,2]
      call  res = get(vec, 1)
L0    print res
```

```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]

var res = x[2+1]
print res
```

Inlining

```
        array vec = [0,0,1,2]
        call  res = get(vec, 1)
L0      print res
```




```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]

var res = x[2+1]
print res
```

Inlining

```
assume
    e, …                                  guards
  else
    F.V.L [x=e, …]                        frame synthesis
    (F.V.L x [x=e, …])*
```

Inlining

```
      array vec = [0,0,1,2]
      call  res = get(vec, 1)
L0    print res
```

```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]
       main.V1.L0 res [vec=vec]
var res = x[2+1]
print res
```

Inlining

```
        array vec = [0,0,1,2]
        call  res = get(vec, 1)
L0      print res
```

```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]
       main.V1.L0 res [vec=vec]
var res = x[2+1]
print res
```

Inlining

```
        array vec = [0,0,1,2]
        call  res = get(vec, 1)
L0      print res
```

```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]
       main.V1.L0 res [vec=vec]
var res = x[2+1]
print res
```

Inlining

```
        array vec = [0,0,1,2]
        call  res = get(vec, 1)
L0      print res
```

```
array vec = [0,0,1,2]
assume (vec ≠ nil)
  else get.V1.L0 [x=vec,i=1,off=2]
       main.V1.L0 res [vec=vec]
var res = x[2+1]
print res
```

Moving Assume

```
     assume true
       else size.V1.L0 [x=x,i=i,off=2]
L0   branch (x = nil) L2 L1
L1

     return x[2+i]
L2   return nil
```

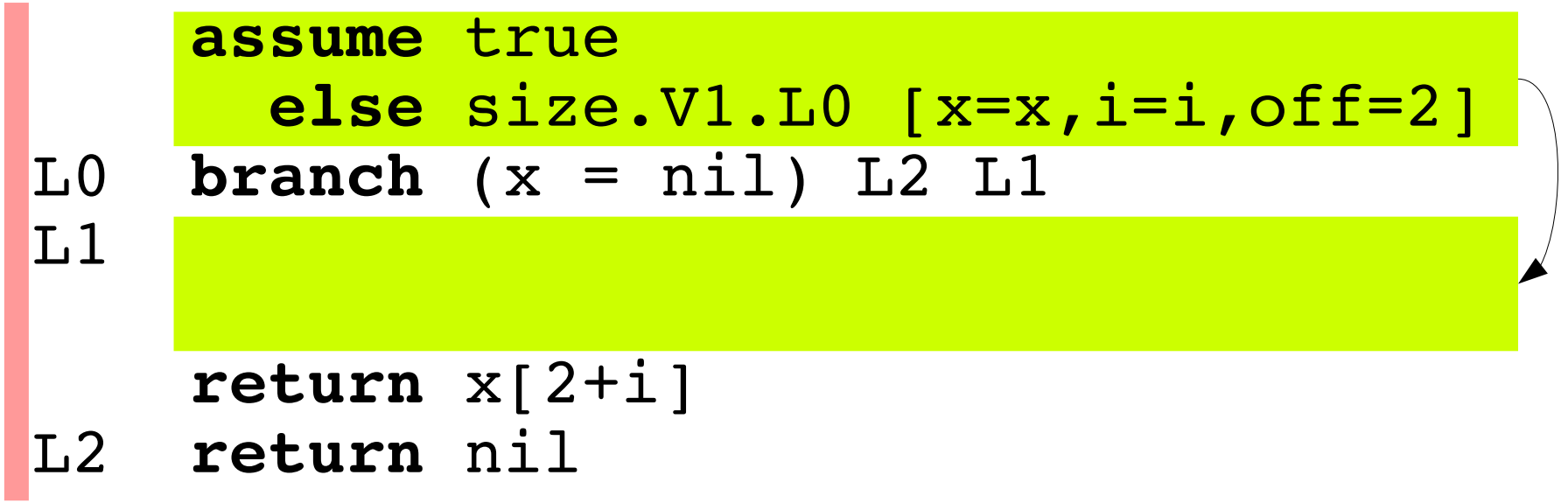
Moving Assume

```
     assume true
       else size.V1.L0 [x=x,i=i,off=2]
L0   branch (x = nil) L2 L1
L1   assume (i = 1)

     return x[2+i]
L2   return nil
```

Moving Assume

```
     assume true
       else size.V1.L0 [x=x,i=i,off=2]
L0   branch (x = nil) L2 L1
L1

     return x[2+i]
L2   return nil
```

Moving Assume



Moving Assume

```
     assume true
       else size.V1.L0 [x=x,i=i,off=2]
L0   branch (x = nil) L2 L1
L1   assume true
       else size.V1.L0 [x=x,i=i,off=2]
     return x[2+i]
L2   return nil
```

Moving Assume

```
     assume true
       else size.V1.L0 [x=x,i=i,off=2]
L0   branch (x = nil) L2 L1
L1   assume (i = 0)
       else size.V1.L0 [x=x,i=i,off=2]
     return x[2+i]
L2   return nil
```

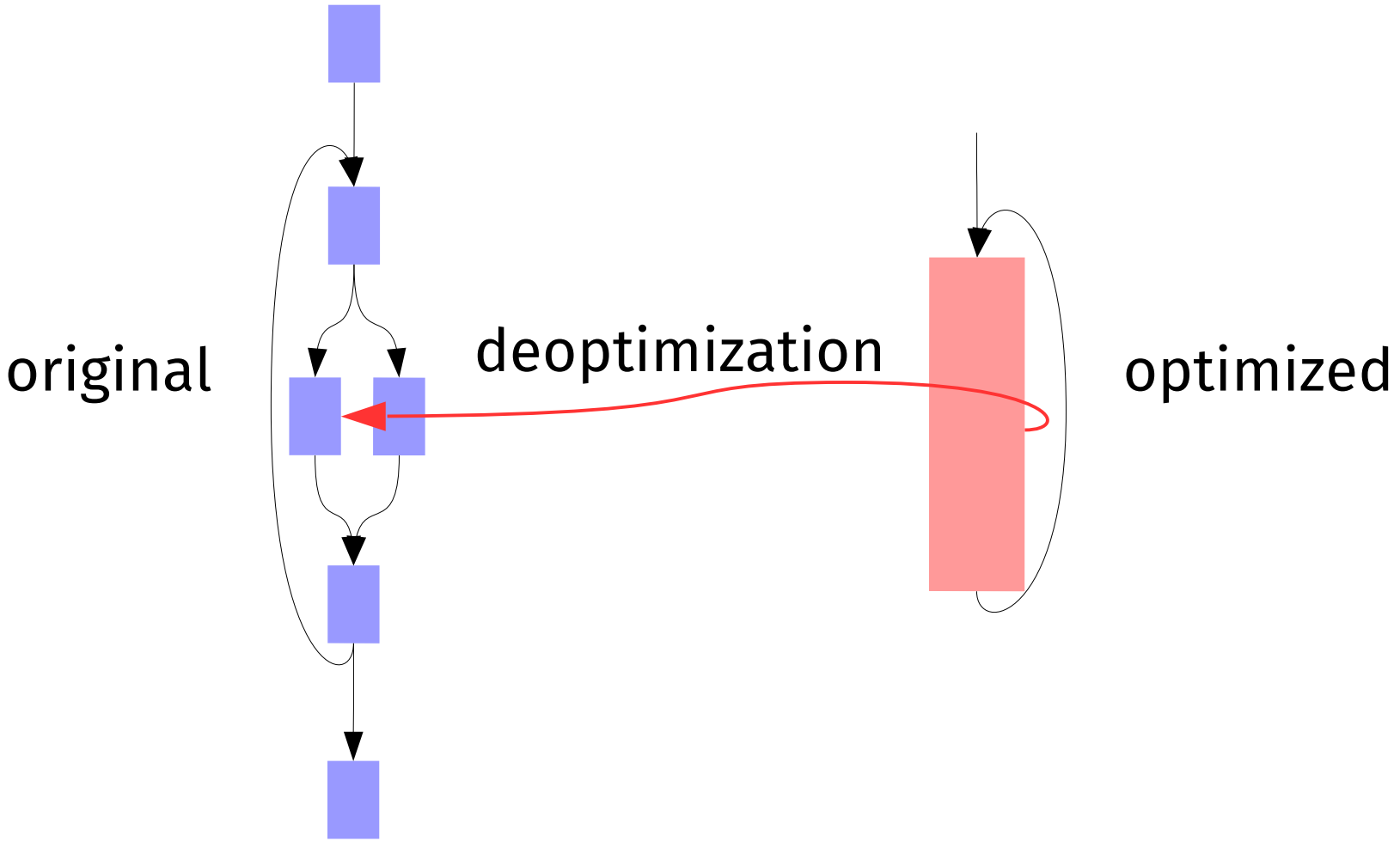
Moving Assume

```
     assume true
       else size.V1.L0 [x=x,i=i,off=2]
L0   branch (x = nil) L2 L1
L1   assume (i = 0)
       else size.V1.L0 [x=x,i=i,off=2]
     return x[2+i]
L2   return nil
```

Correctness

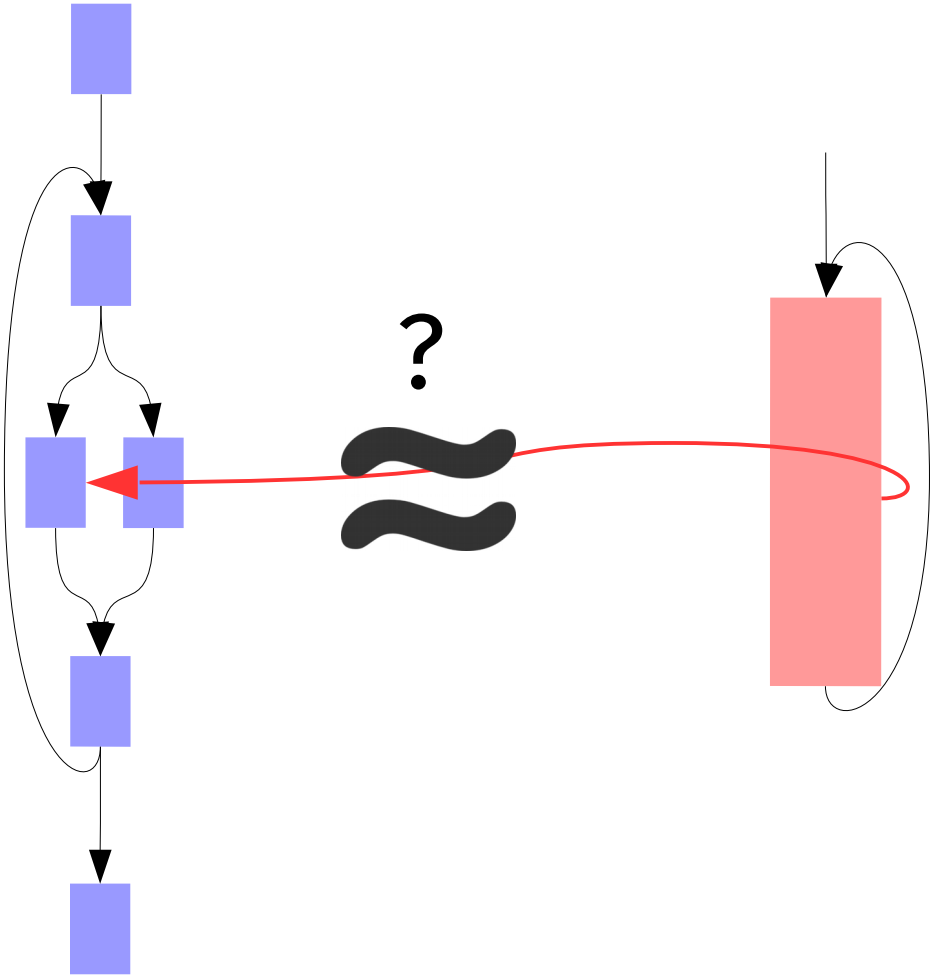
Correctness

Proof Structure



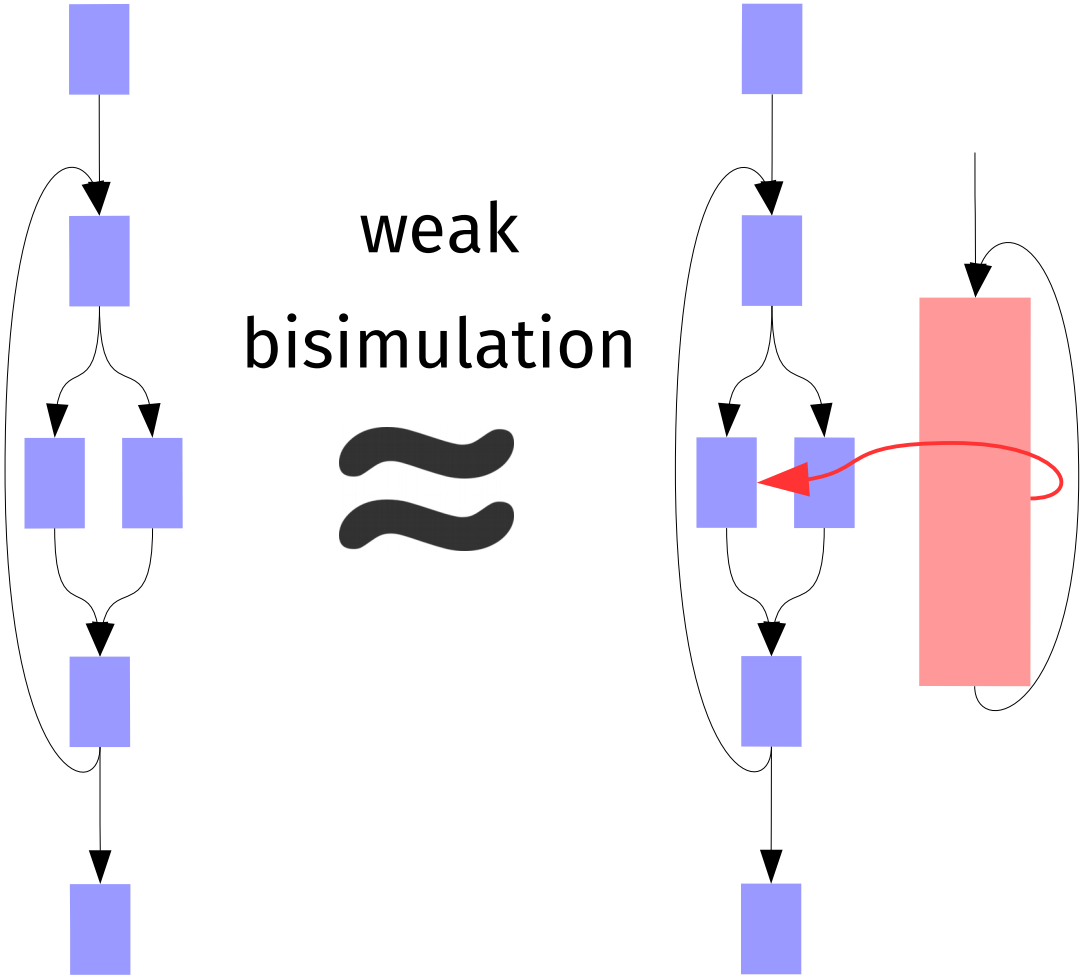
Correctness

Proof Structure

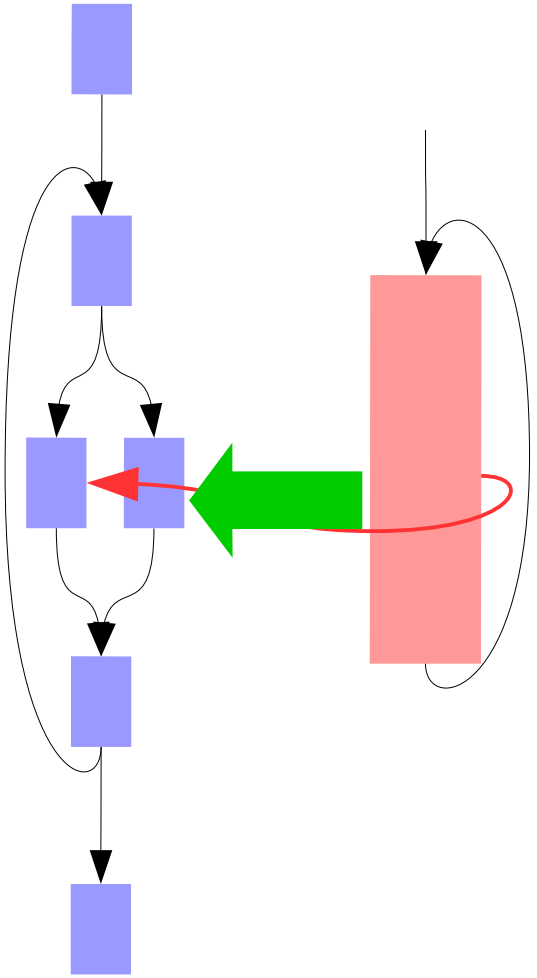


Correctness

Proof Structure



Correctness Assumption Transparency

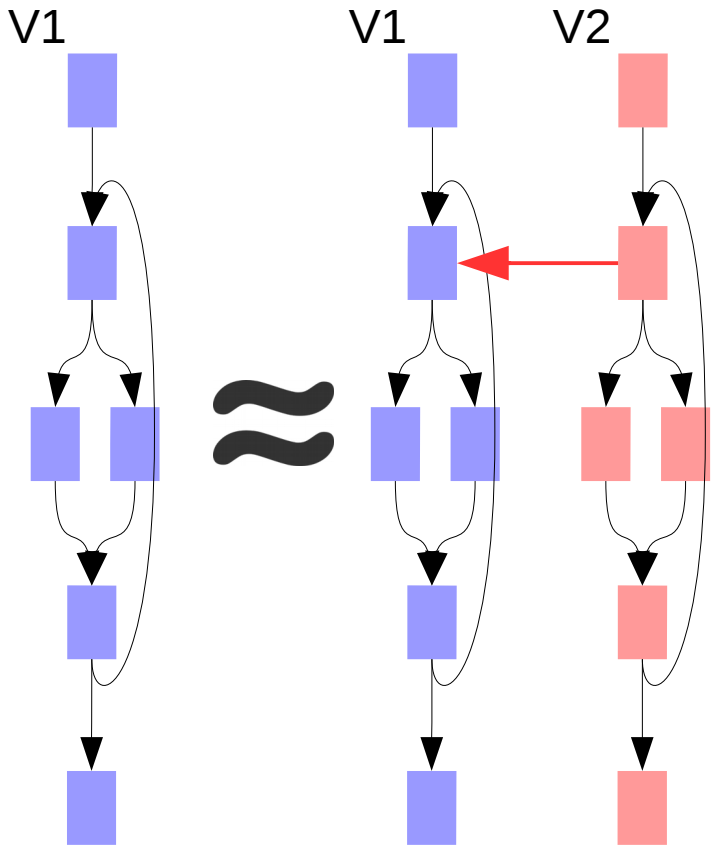


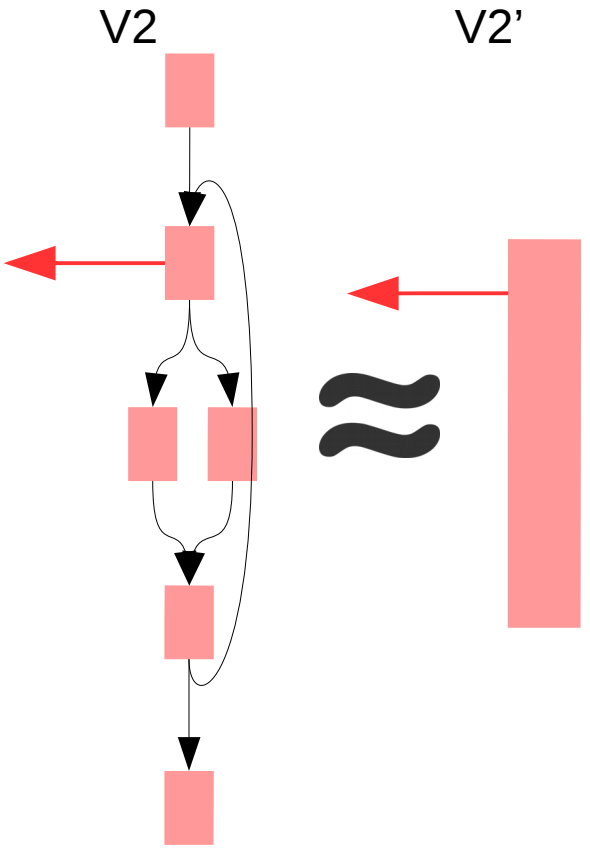
Deoptimizing even if guard holds

does not alter behavior

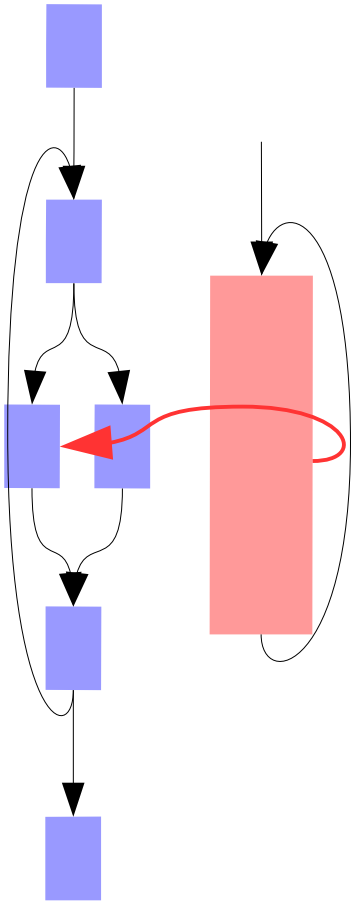
Correctness

Division of Labour





Takeaways



Deoptimization is part of IR semantics

Reifying metadata simplifies reasoning

Optimizations can be easily combined with deoptimization, given some consideration