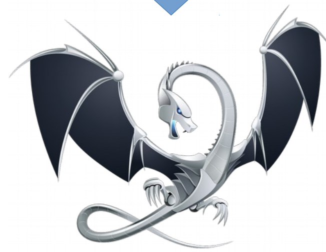
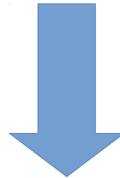
Compilers for Uncooperative Languages

Olivier Flückiger

Batch Compiler

print “HelloWorld”



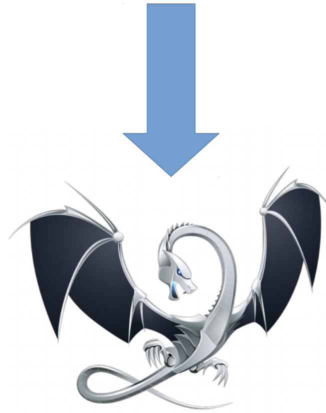


1010101010

Batch Compiler

print “HelloWorld”

compile-time



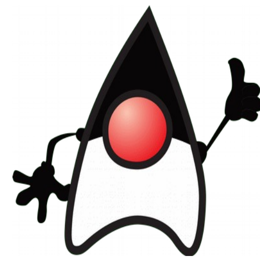
run-time

1010101010

JIT Compiler

print “HelloWorld”

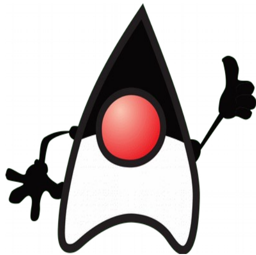
push “hello world”
call “print”



JIT Compiler

print “HelloWorld”

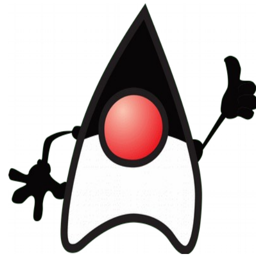
push “hello world”
call “print”

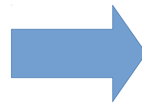


JIT Compiler

print "HelloWorld"

push "hello world"
call "print"

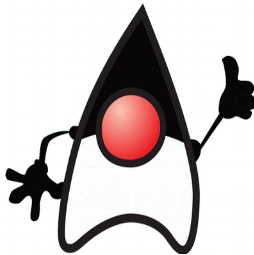


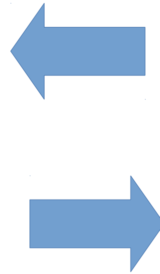


JIT Compiler

print “HelloWorld”

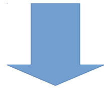
push “hello world”
call “print”

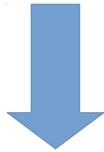




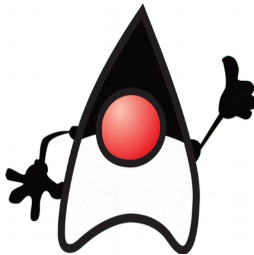
JIT Compiler

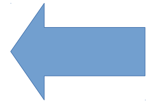
print “HelloWorld”

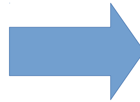


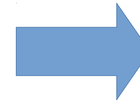


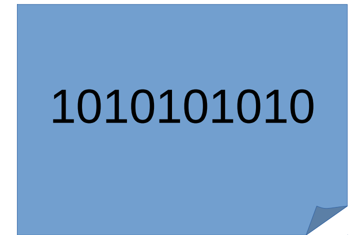
push “hello world”
call “print”









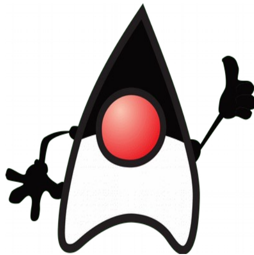


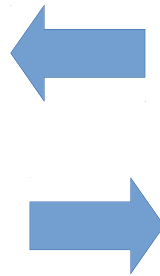
1010101010

JIT Compiler

print “HelloWorld”

push “hello world”
call “print”

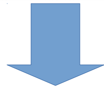




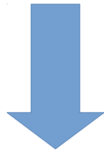
1010101010

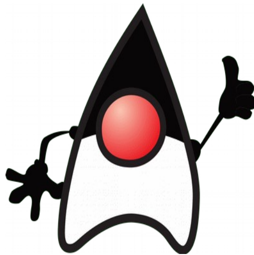
JIT Compiler

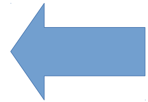
print “HelloWorld”

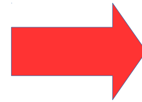


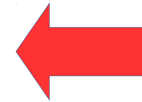
push “hello world”
call “print”











1010101010

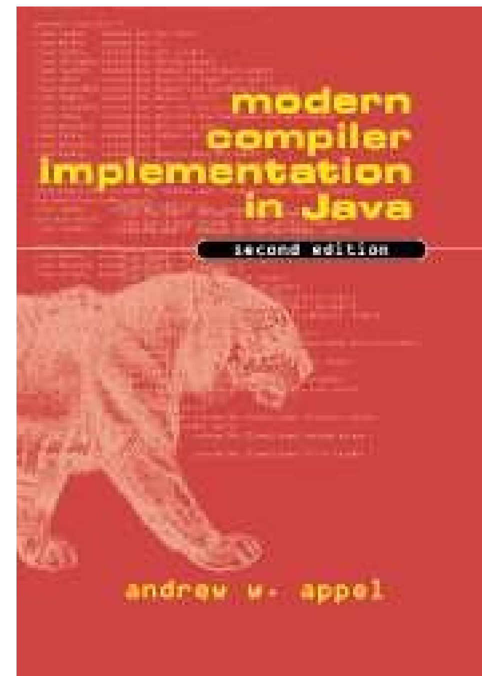
Some Basic Tricks in the Book

Speculative Optimizations

Case Study R

Basic Tricks

(not in this book



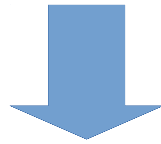
)

Inline Caches

```
function(var s)
    s.width()
```

function (var shape)

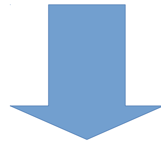
shape.width()



```
m = shape.proto.lookup(msg)
invoke(shape, m)
```

function (var shape)

shape.width()



m = **shape.proto**.lookup(msg)

invoke(shape, m)

Published in ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991.

Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Urs Hölzle
Craig Chambers
David Ungar†

Computer Systems Laboratory, Stanford University, Stanford, CA 94305
{urs,craig,ungar}@self.stanford.edu

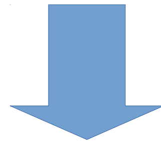
Abstract: *Polymorphic inline caches* (PICs) provide a new way to reduce the overhead of polymorphic message sends by extending inline caches to include more than one cached lookup result per call site. For a set of typical object-oriented SELF programs, PICs achieve a median speedup of 11%.

As an important side effect, PICs collect type information by recording all of the receiver types actually used at a given call site. The compiler can exploit this type information to generate better code when *recompiling* a method. An experimental version of such a system achieves a median speedup of 27% for our set of SELF programs, reducing the number of non-inlined message sends by a factor of two.

Implementations of dynamically-typed object-oriented languages have been limited by the paucity of type information available to the compiler. The abundance of the type information provided by PICs suggests a new compilation approach for these languages, *adaptive compilation*. Such compilers may succeed in generating very efficient code for the time-critical parts of a program without incurring distracting compilation pauses.

function (var shape)

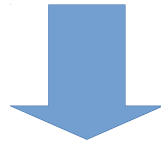
shape.width()



inlineCacheFail(shape, “width”)

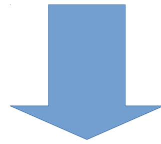
function (var shape)

shape.width()



inlineCach[illegible]ape, “width”)

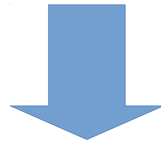
```
function (var shape)
    shape.width()
```



```
If obj.proto == Rectangle
    Rectangle::width(shape)
else
    inlineCacheFail(shape, “width”)
```

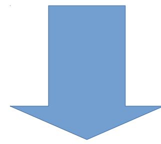
function (var shape)

shape.width()



```
If obj.proto == Rectangle
    Rectangl… idth(shape)
else
    inlineCacheFail(shape, “width”)
```

```
function (var shape)
    shape.width()
```



```
If obj.proto == Rectangle
    Rectangle::width(shape)
elif obj.proto == Circle
    Circle::width(shape)
else
    inlineCacheFail(shape, “width”)
```

Hidden Classes

```
point = {}
point.x = 1
point.y = 2
```

```
point = {}
point.x = 1
if (hasY) {
    point.y = 2
}
```

```
point = No static object layout
point.x = 1
if (hasY) {
    point.y = 2
}
```

An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*

Craig Chambers, David Ungar, and Elgin Lee

Center for Integrated Systems, Stanford University

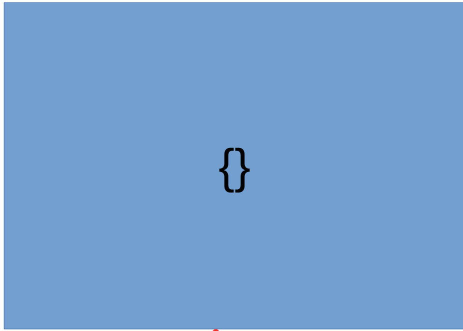
self@self.stanford.edu

Abstract

We have developed and implemented techniques that double the performance of dynamically-typed object-oriented languages. Our SELF implementation runs

1. Introduction

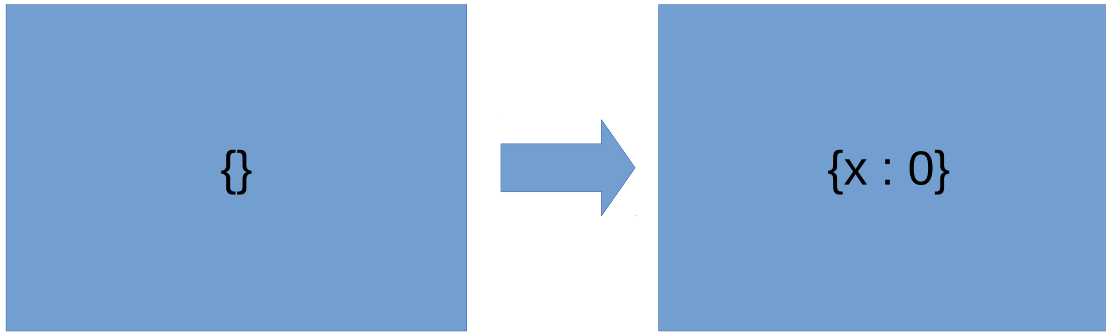
SELF [US87] is a dynamically-typed object-oriented language inspired by the Smalltalk-80** language [GR83]. Like Smalltalk, SELF has no type declarations



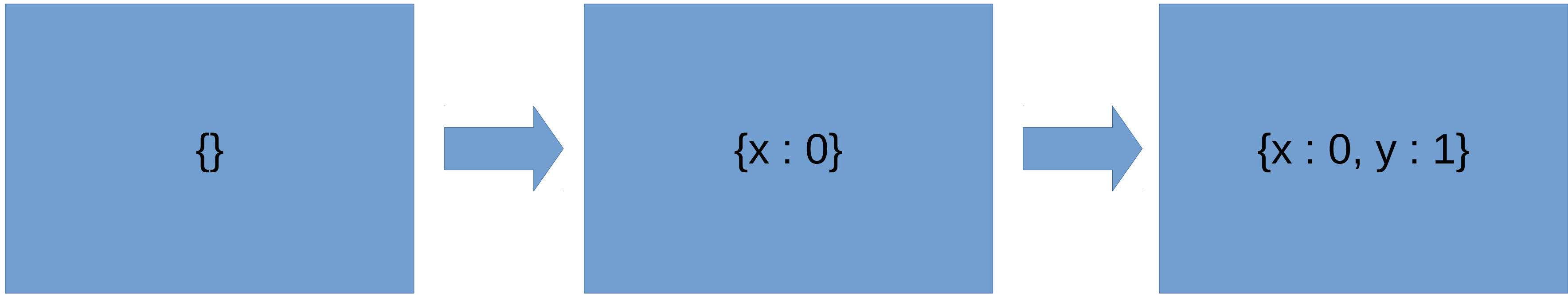
point = {}

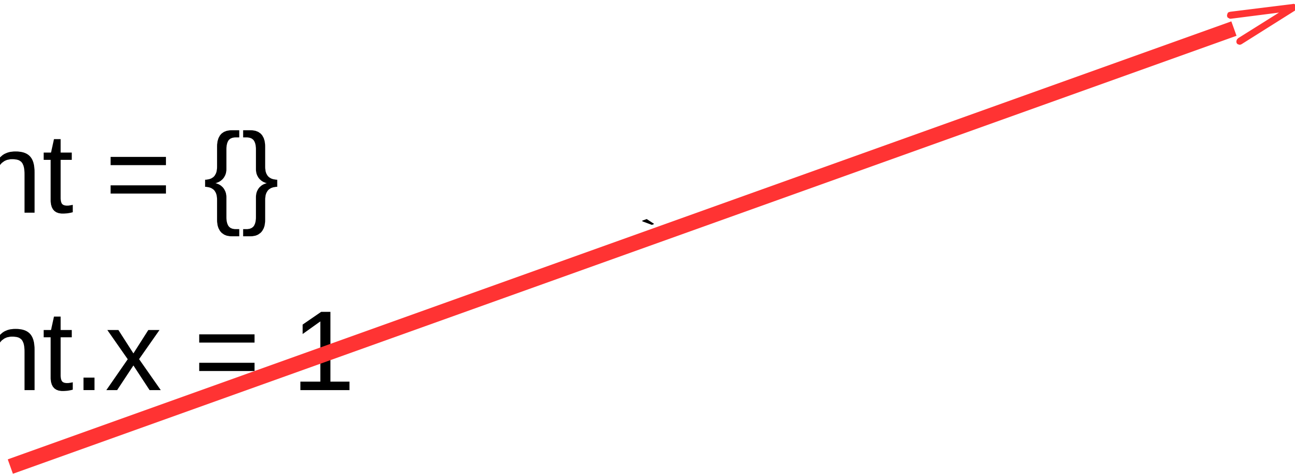
point.x = 1

point.y = 2




```
point = {}
point.x = 1
point.y = 2
```





```
point = {}
point.x = 1
point.y = 2
```

Without Maps

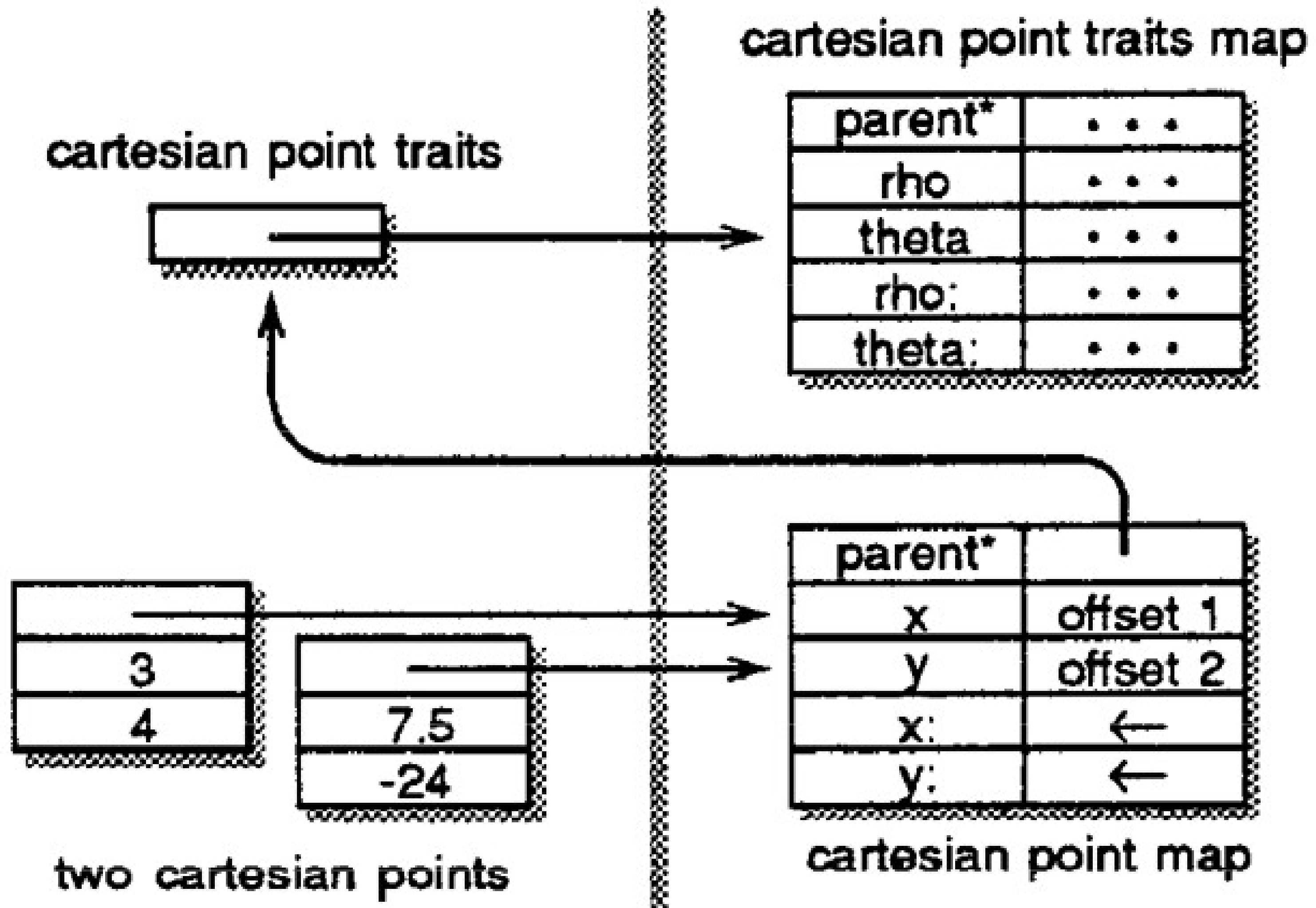
cartesian point traits

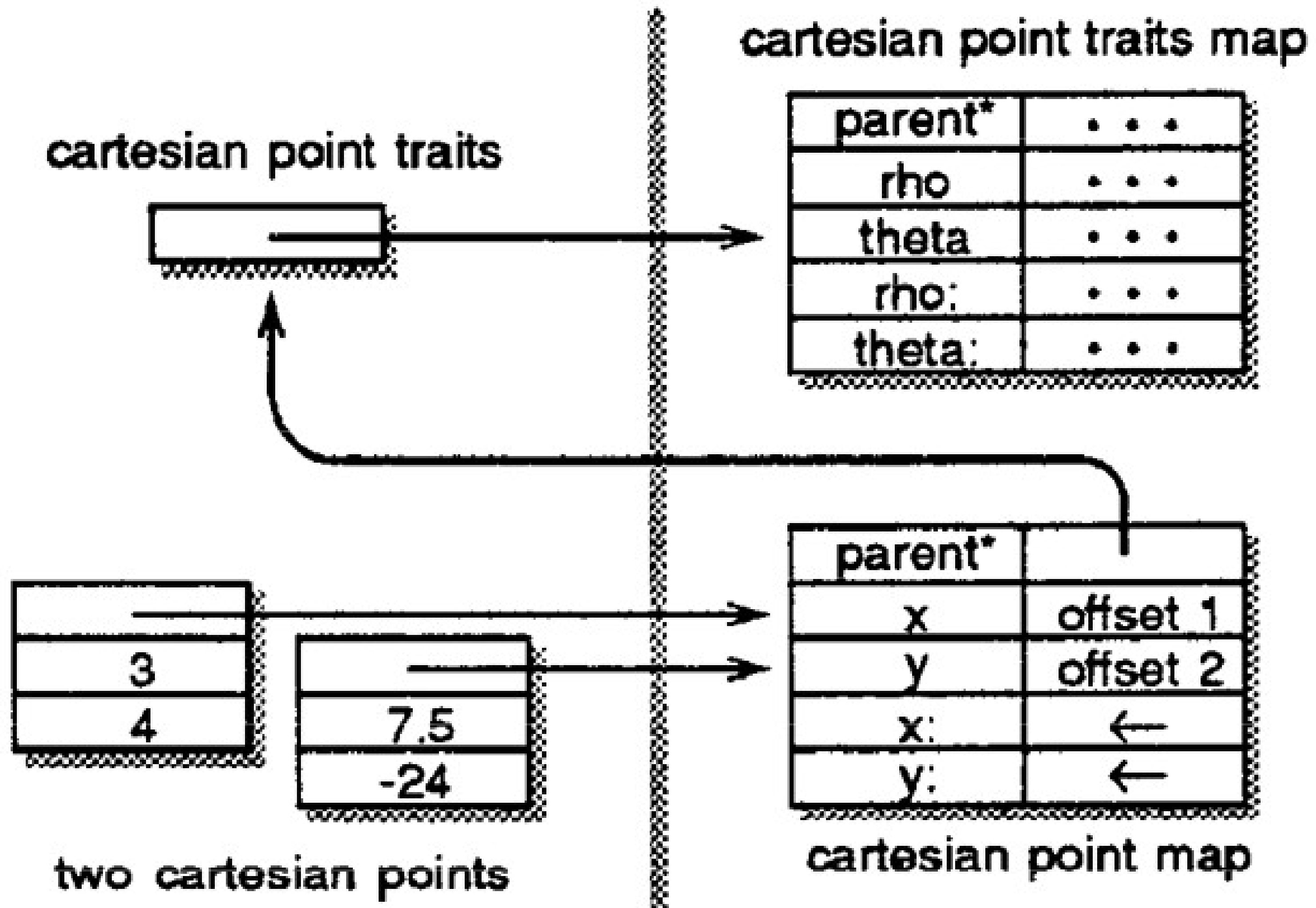
parent*	. . .
rho	. . .
theta	. . .
rho:	. . .
theta:	. . .

parent*	
x	3
y	4
x:	←
y:	←

parent*	
x	7.5
y	-24
x:	←
y:	←

two cartesian points


With Maps
cartesian point traits map
parent*
rho
theta
rho:
theta:
cartesian point traits
parent*
x
y
x:
y:
offset 1
offset 2
3
4
7.5
-24
two cartesian points
cartesian point map



With Maps
cartesian point traits map
parent*
rho
theta
rho:
theta:
cartesian point traits
3
4
7.5
-24
parent*
x
offset 1
y
offset 2
x:
y:
two cartesian points
cartesian point map

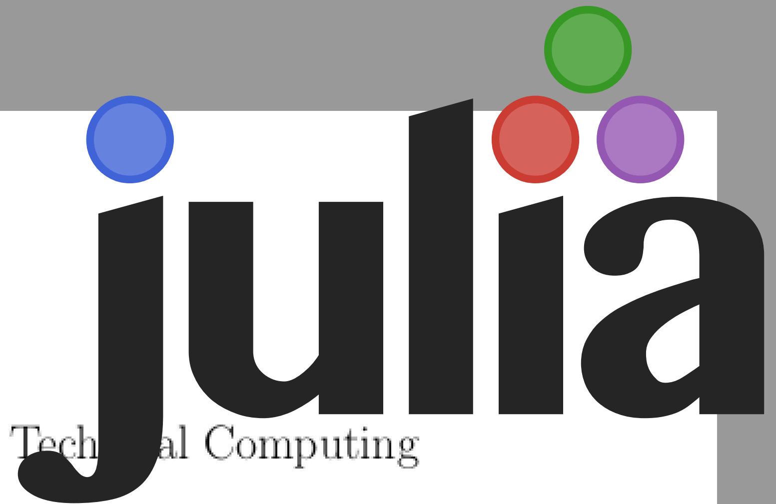
Specialization

```
function add(a, b)
    return a+b
```

```
function add(a, b)
   return a+b

add(1, 2)
```

```
function add(a, b)
    return a+b

add(1, 2)
add(1.1, 2.2)
```

```
function add(a, b)
   return a+b

add(1, 2)
add(1.1, 2.2)
add(“1”, “2”)
```



Julia: A Fast Dynamic Language for Tech[illegible]al Computing

Jeff Bezanson* MIT | Stefan Karpinski† MIT | Viral B. Shah‡ | Alan Edelman§ MIT

September 25, 2012

:1209.5145v1 [cs.PL] 24 Sep 2012

Abstract

Dynamic languages have become popular for scientific computing. They are generally considered highly productive, but lacking in performance. This paper presents Julia, a new dynamic language for technical computing, designed for performance from the beginning by adapting and extending modern programming language techniques. A design based on generic functions and a rich type system simultaneously enables an expressive programming model and successful type inference, leading to good performance for a wide range of programs. This makes it possible for much of Julia's library to be written in Julia itself, while also incorporating best-of-breed C and Fortran libraries.

1 Introduction

Convenience is winning. Despite advances in compiler technology and execution for high-performance computing, programmers continue to prefer high-level dynamic languages for algorithm development and data analysis in applied math, engineering, and the sciences. High-level environments such as MATLAB®, Octave [26], R [18], SciPy [29], and SciLab [17] provide greatly increased convenience and productivity. However, C and Fortran remain the gold standard languages for computationally-intensive problems because high-level dynamic languages still lack sufficient performance. As a result, the most challenging areas of technical computing have benefited the least from the increased abstraction and productivity offered by

```
function add(a, b)
   return a+b

add(1, 2)
add(1.1, 2.2)
add(“1”, “2”)
```

```
function add(a, b)
  return a+b


add(1, 2)
add(1.1, 2.2)
add("1", "2")
```

```
function add(a, b)
   return a+b

add(1, 2)
add(1.1, 2.2)
add("1", "2")
```

add : i64, i64 → i64

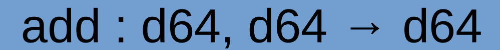
add : d64, d64 → d64

```
function add(a, b)
   return a+b
```

add(1, 2)

add(1.1, 2.2)

add(“1”, “2”)



+ simple, no deoptimization needed

- needs complex typesystem

Tracing

Trace-based Just-in-Time Type Specialization for Dynamic Languages

Andreas Gal[*+], Brendan Eich[*], Mike Shaver[*], David Anderson[*], David Mandelin[*],
Mohammad R. Haghighat[$], Blake Kaplan[*], Graydon Hoare[*], Boris Zbarsky[*], Jason Orendorff[*],
Jesse Ruderman[*], Edwin Smith[#], Rick Reitmaier[#], Michael Bebenita[+], Mason Chang[+#], Michael Franz[+]

Mozilla Corporation[*]
{gal,brendan,shaver,danderson,dmandelin,mrbkap,graydon,bz,jorendorff,jruderman}@mozilla.com

Adobe Corporation[#]
{edwsmith,rreitmai}@adobe.com

Intel Corporation[$]
{mohammad.r.haghighat}@intel.com

University of California, Irvine[+]
{mbebenit,changm,franz}@uci.edu

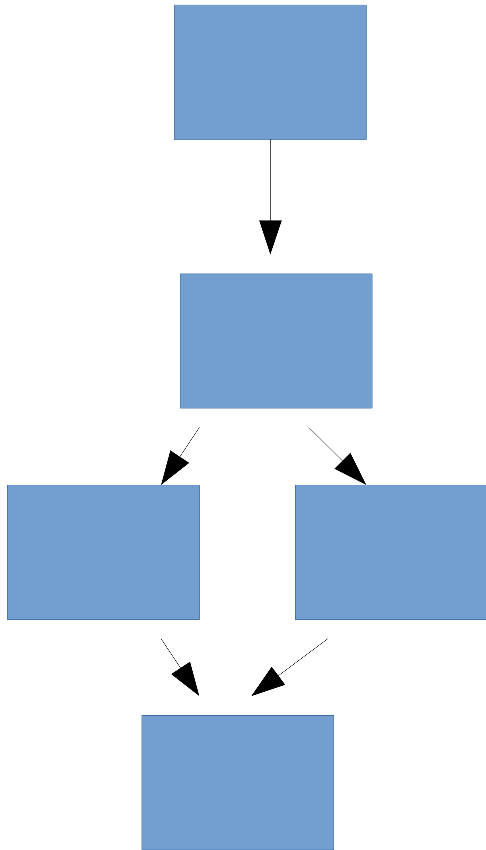
Abstract

Dynamic languages such as JavaScript are more difficult to compile than statically typed ones. Since no concrete type information is available, traditional compilers need to emit generic code that can handle all possible type combinations at runtime. We present an alternative compilation technique for dynamically-typed languages that identifies frequently executed loop traces at run-time and then generates machine code on the fly that is specialized for the actual dynamic types occurring on each path through the loop. Our method provides cheap inter-procedural type specialization, and an elegant and efficient way of incrementally compiling lazily discovered alternative paths through nested loops. We have implemented

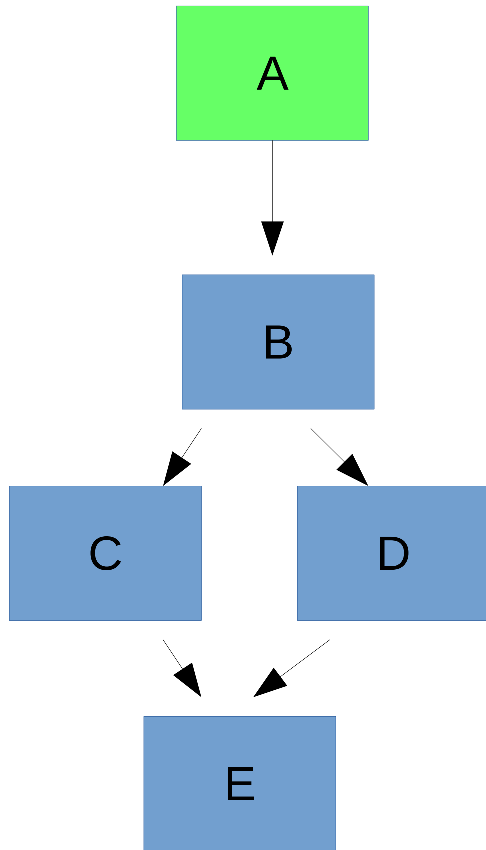
and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. In this domain, in order to provide a fluid user experience and enable a new generation of applications, virtual machines must provide a low startup time and high performance.

Compilers for statically typed languages rely on type information to generate efficient machine code. In a dynamically typed programming language such as JavaScript, the types of expressions may vary at runtime. This means that the compiler can no longer easily transform operations into machine instructions that operate on one specific type. Without exact type information, the compiler must emit slower generalized machine code that can deal with all

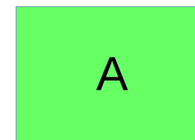
interpreter



interpreter

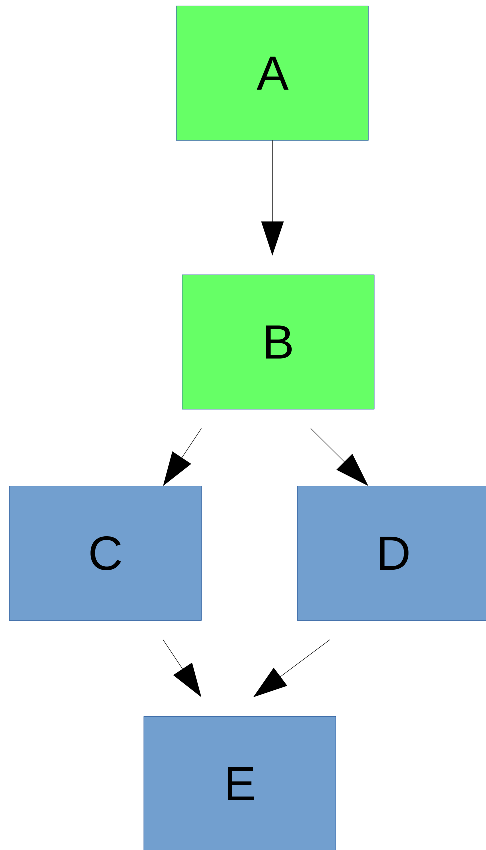


native



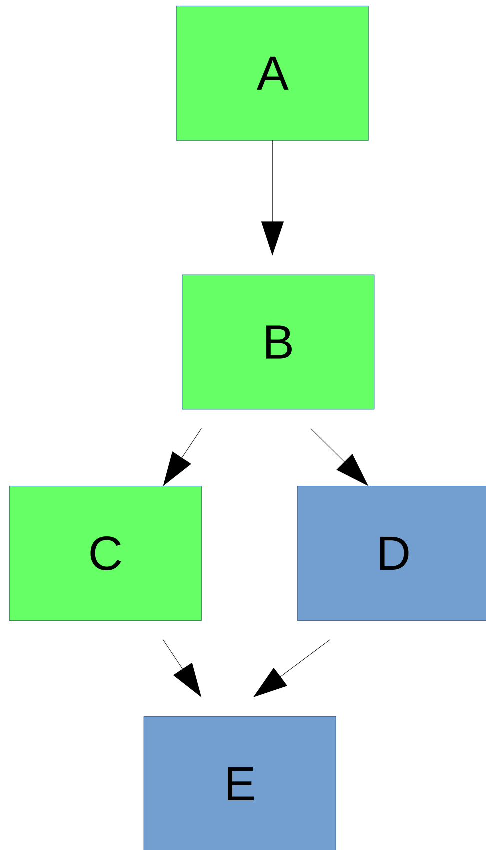
interpreter

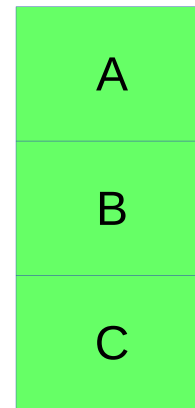
native



interpreter

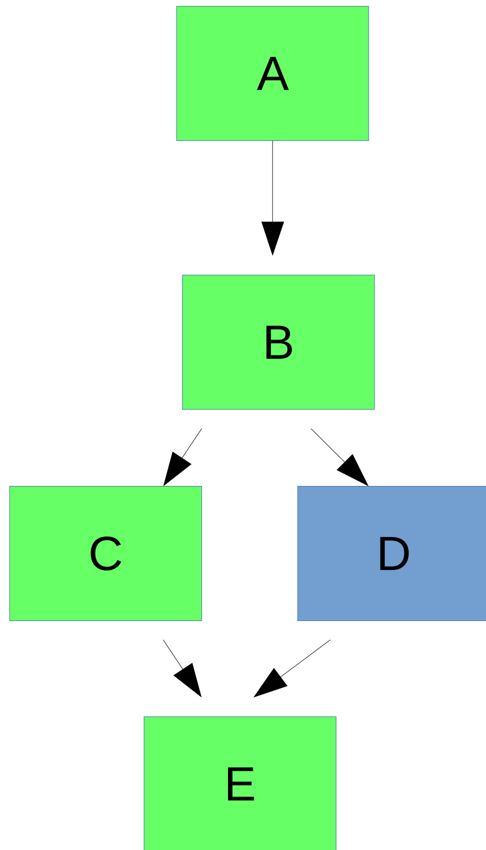
native

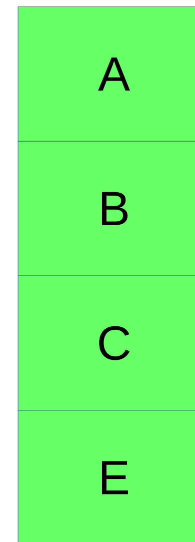




interpreter

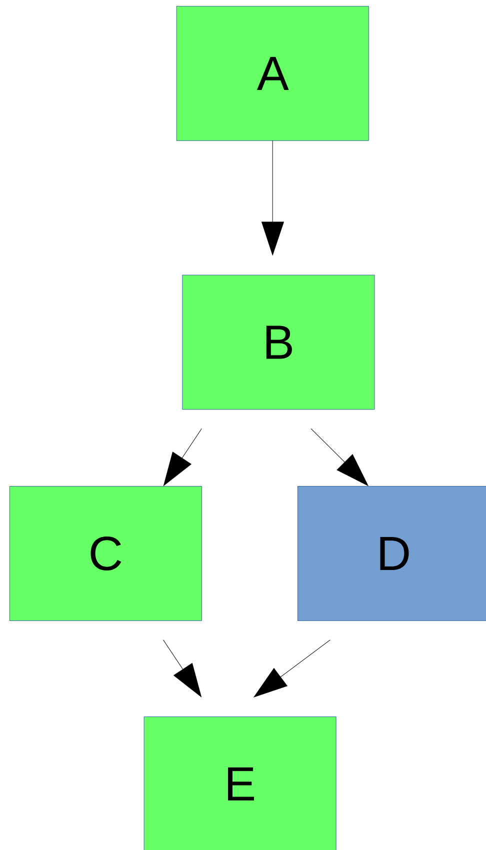
native

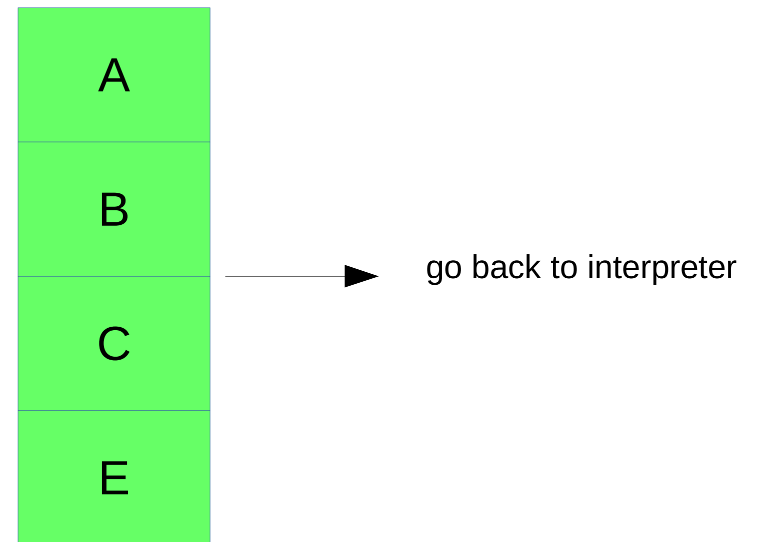




interpreter

native





+ very simple to implement

+ good performance for stable code

+ loop unrolling for free

+ very simple to implement

+ good performance for stable code

+ loop unrolling for free

- traces hard to optimize further

- path explosion

Speculative Optimizations

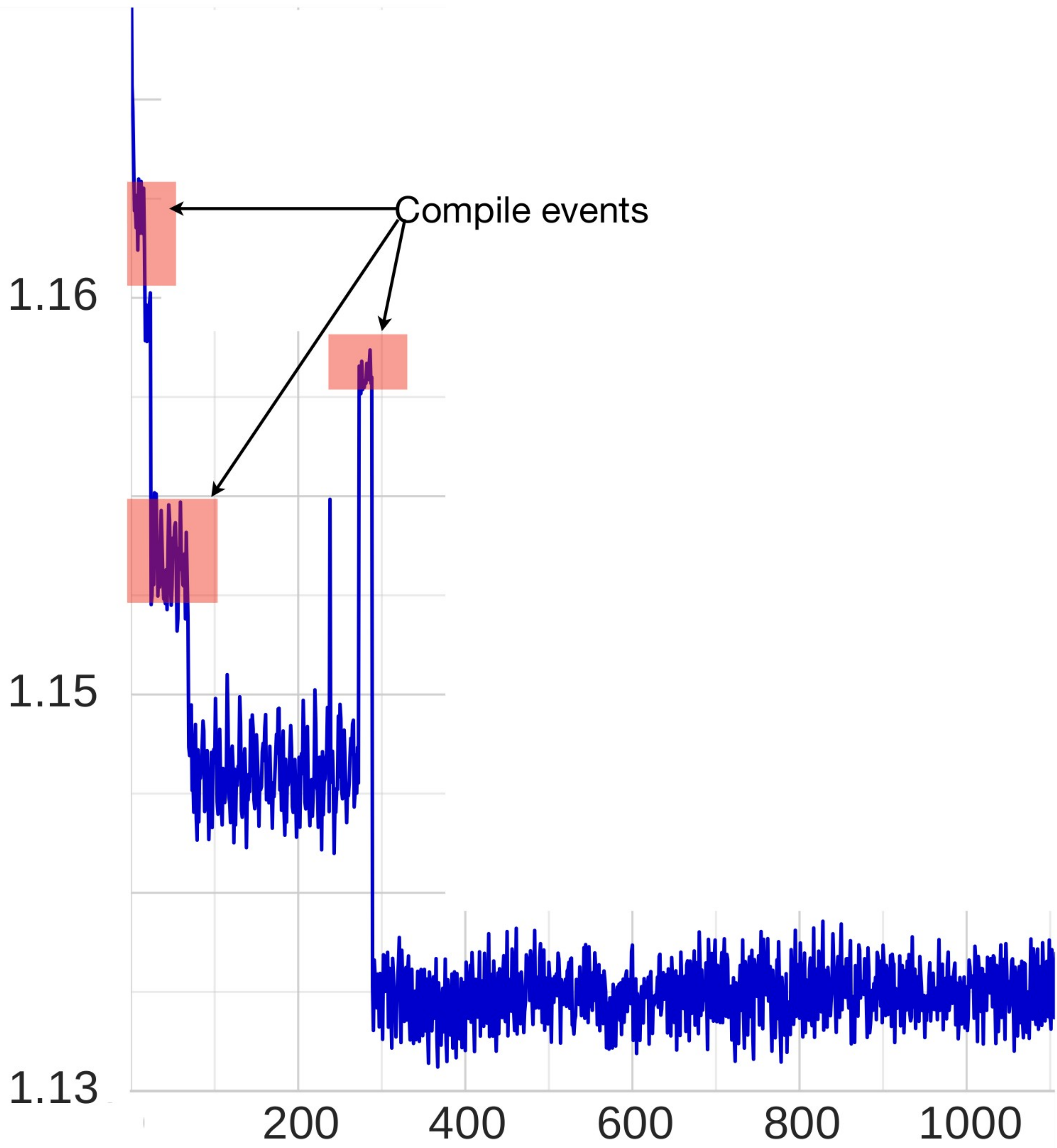
Case Study: R

Speculation

```
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) {
    …
  …
}
```



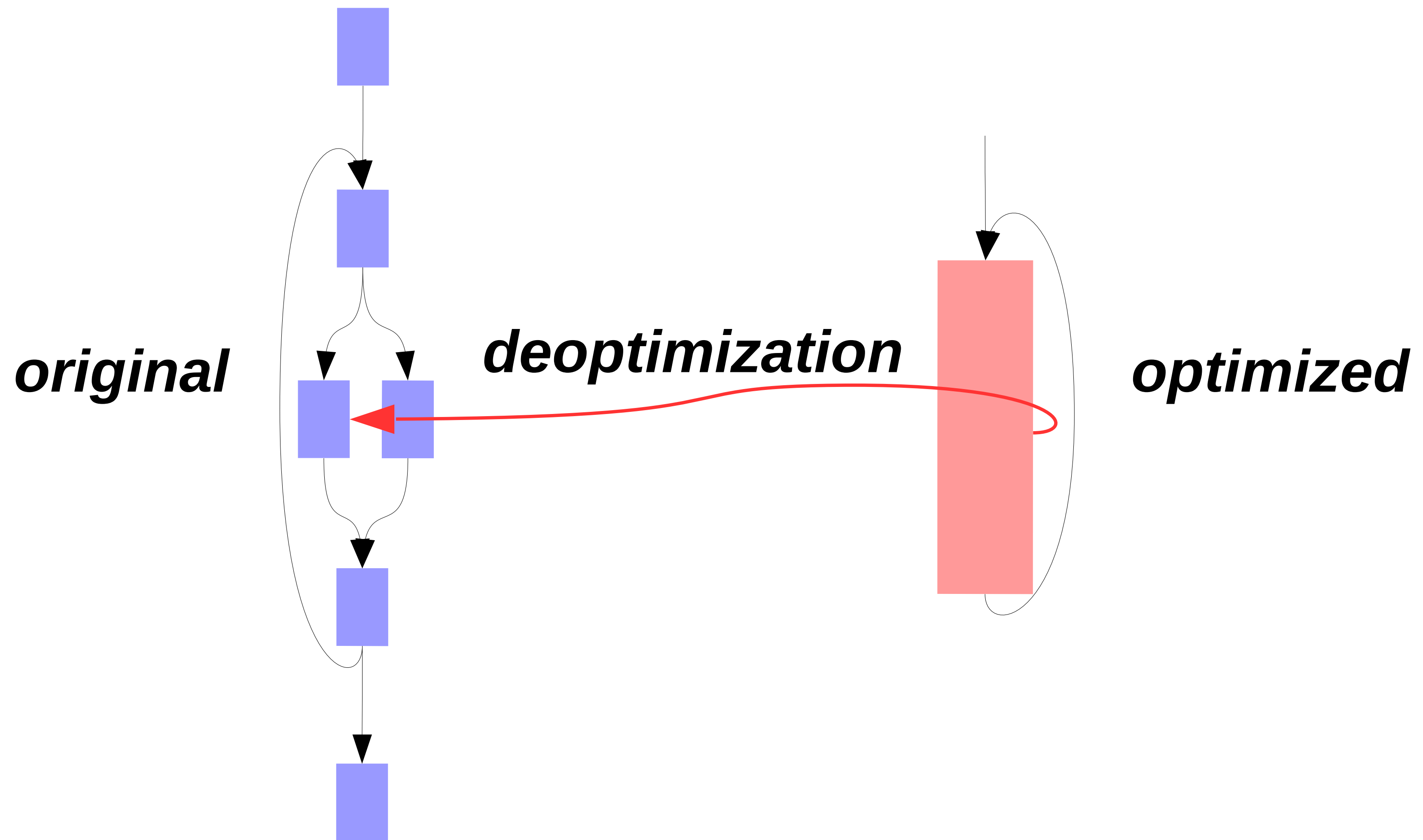
Fasta JS benchmark,
V8, Linux, i7-*4790*

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

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



    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
}
```

Correctness



Me: how can this be true?

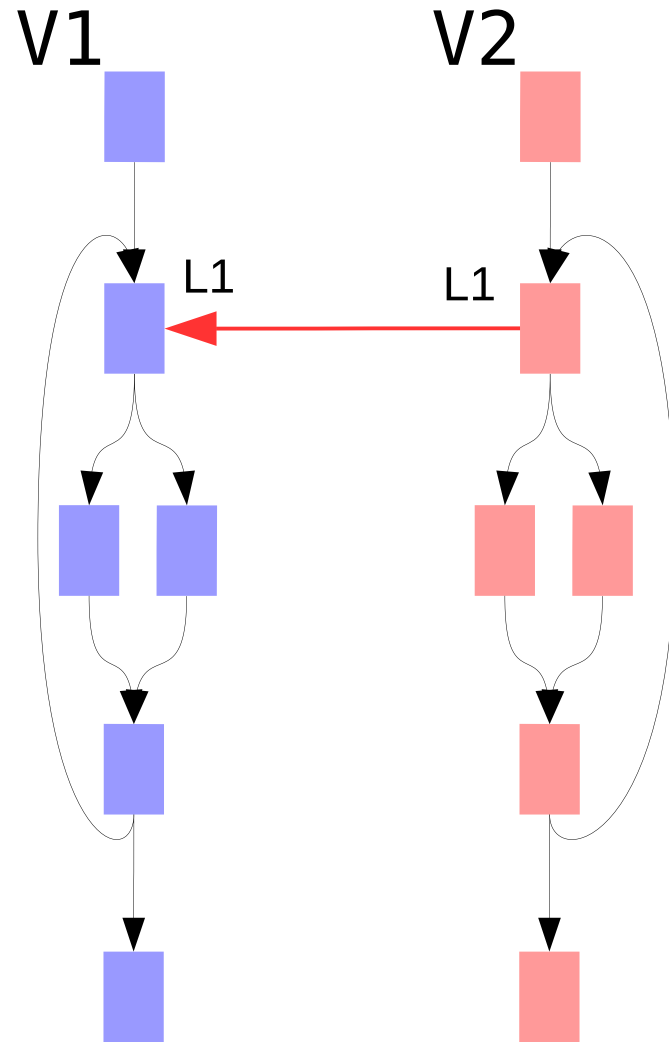
Correctness of Speculative Optimizations with Dynamic Deoptimization

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

Northeastern University,
INRIA, and CVUT

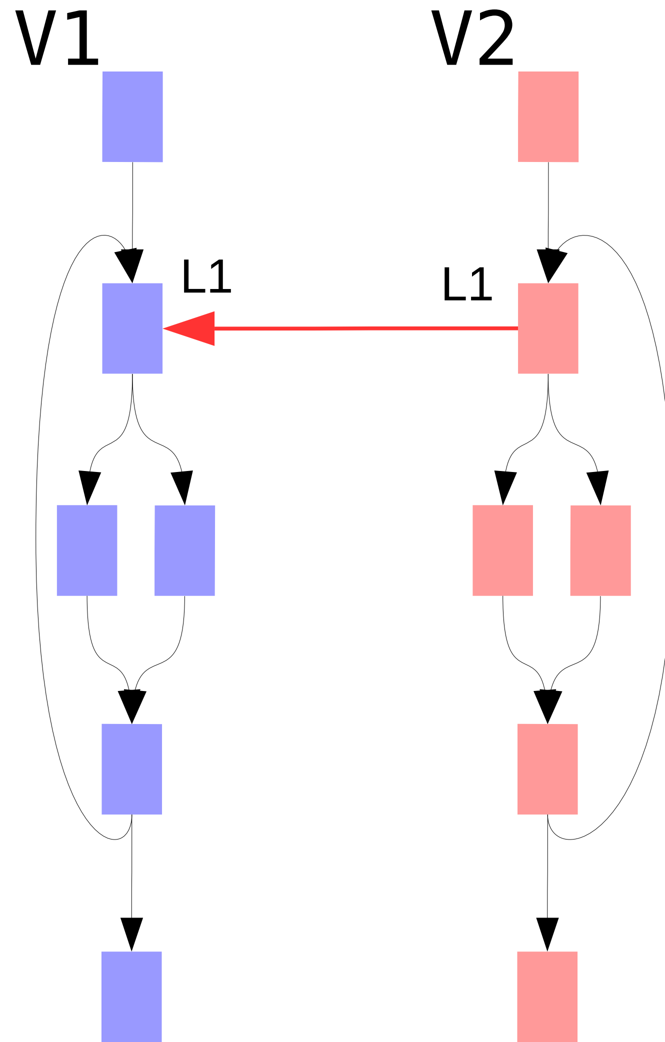
Writing a JIT

Overview



1. Create identical copy

Overview

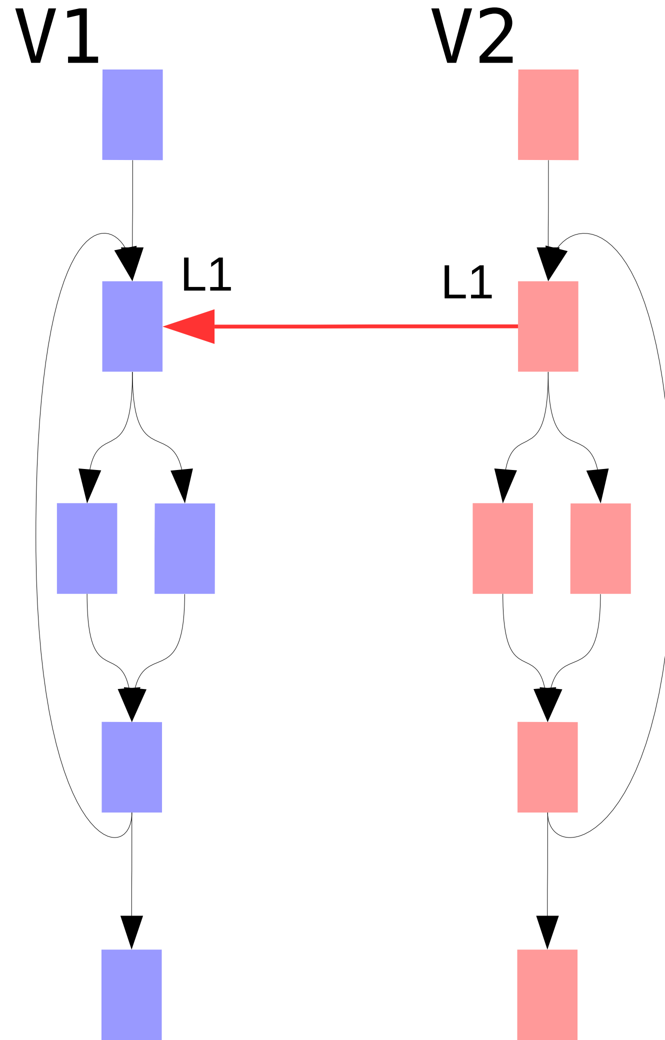


1. Create identical copy

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

```
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

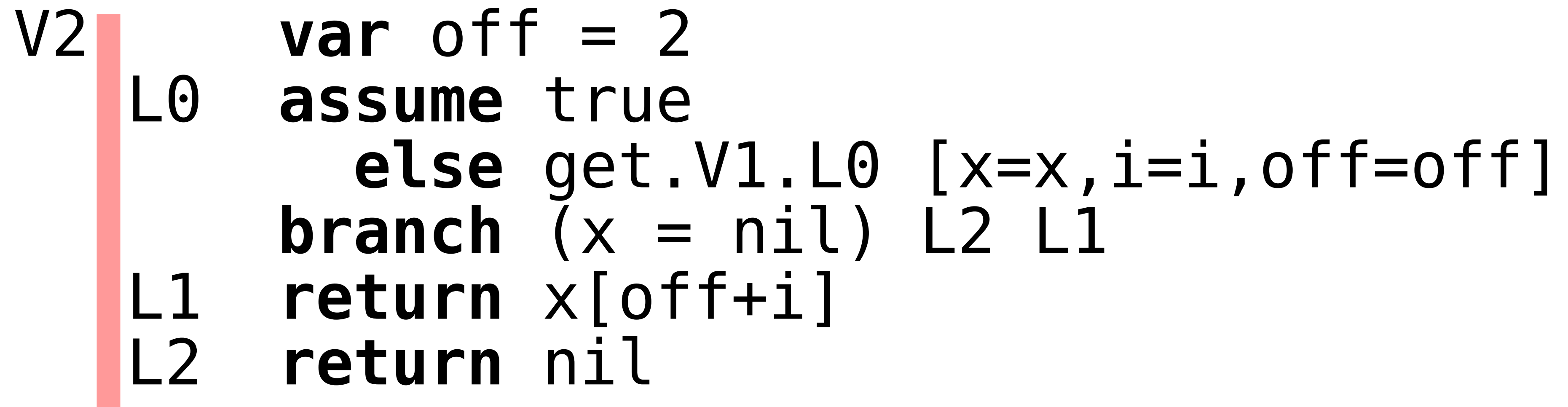
```
get(x, i)

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
```

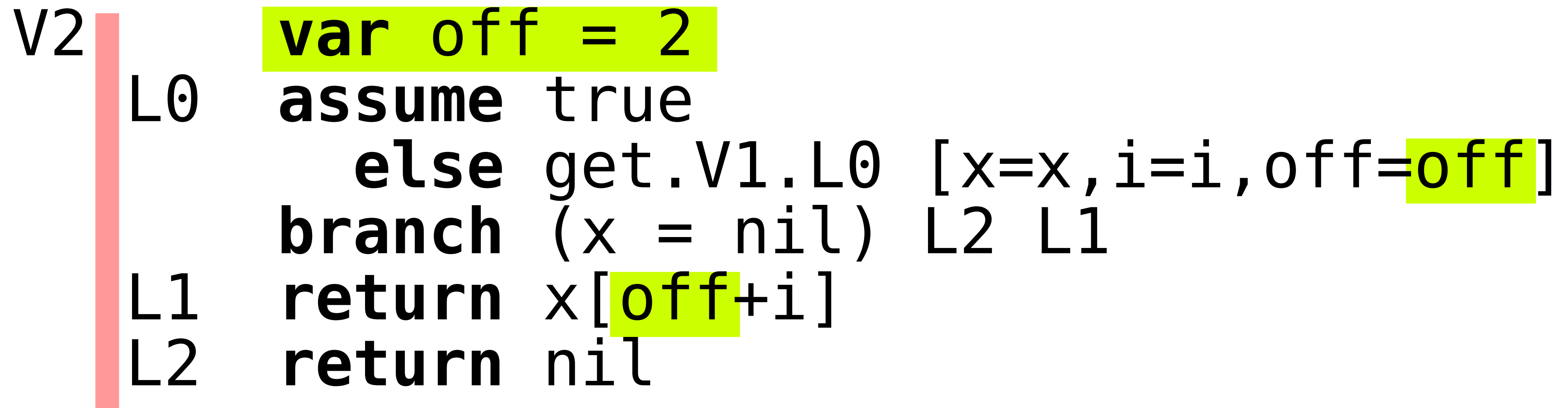
Adding Empty Assume

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



Constant Propagation

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



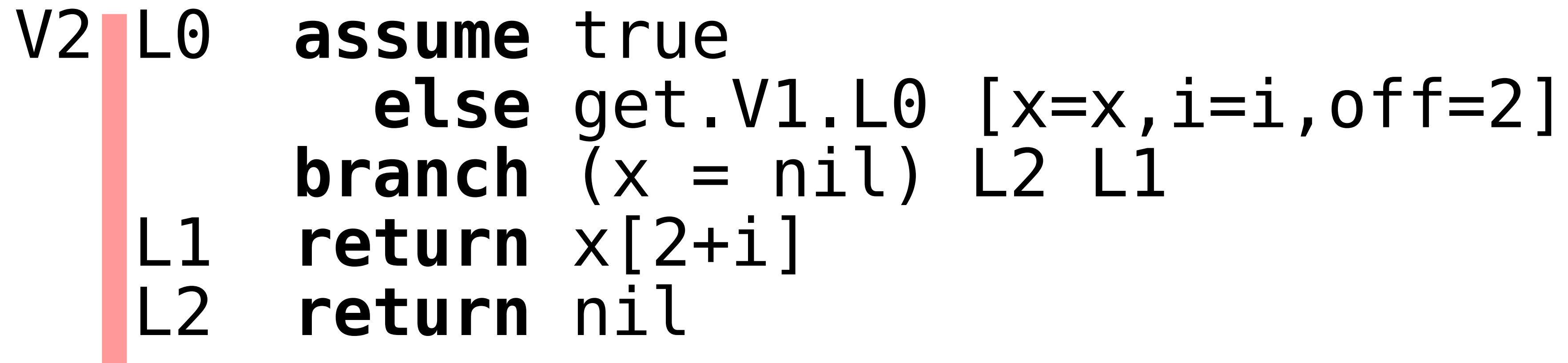
Constant Propagation

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

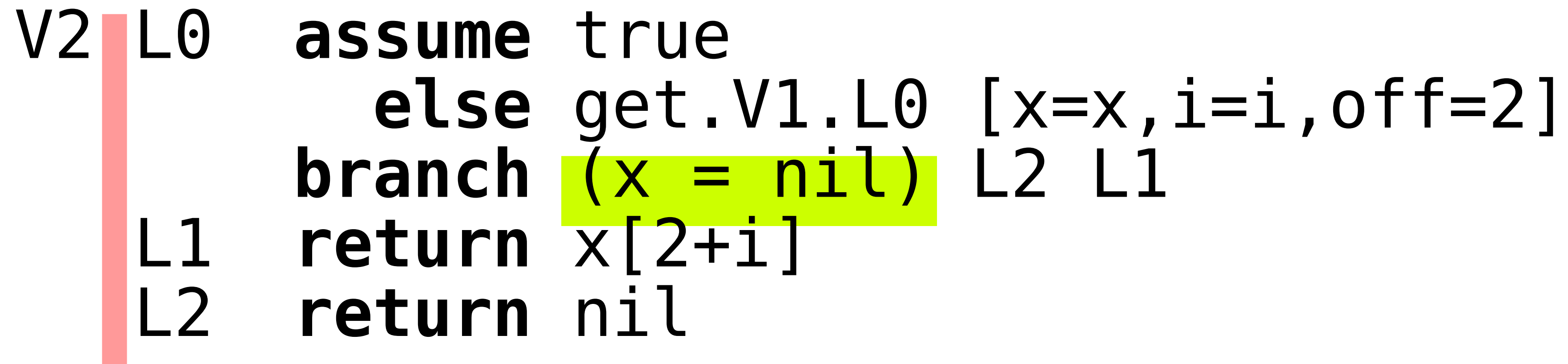
```
 V1        …
```

```
 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
```

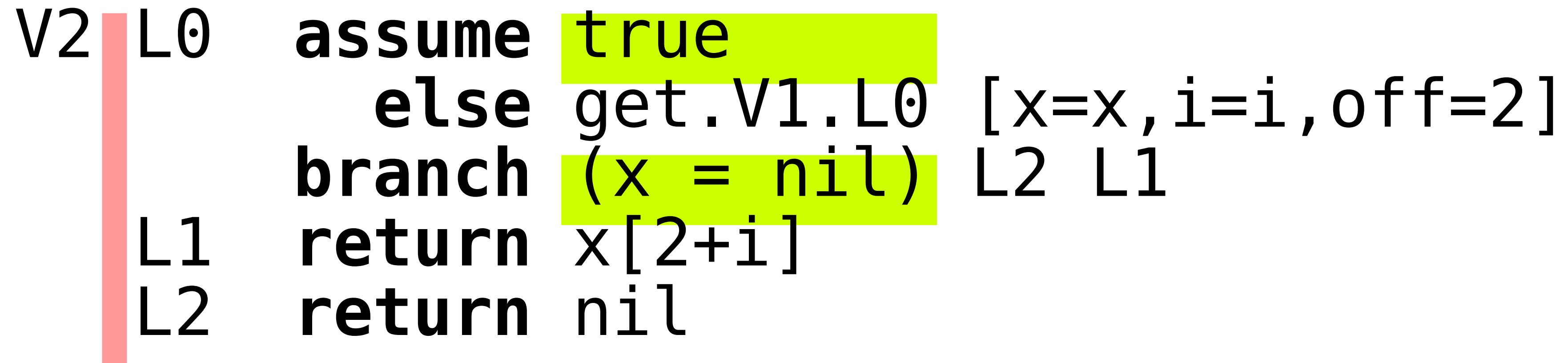
Speculation



Speculation

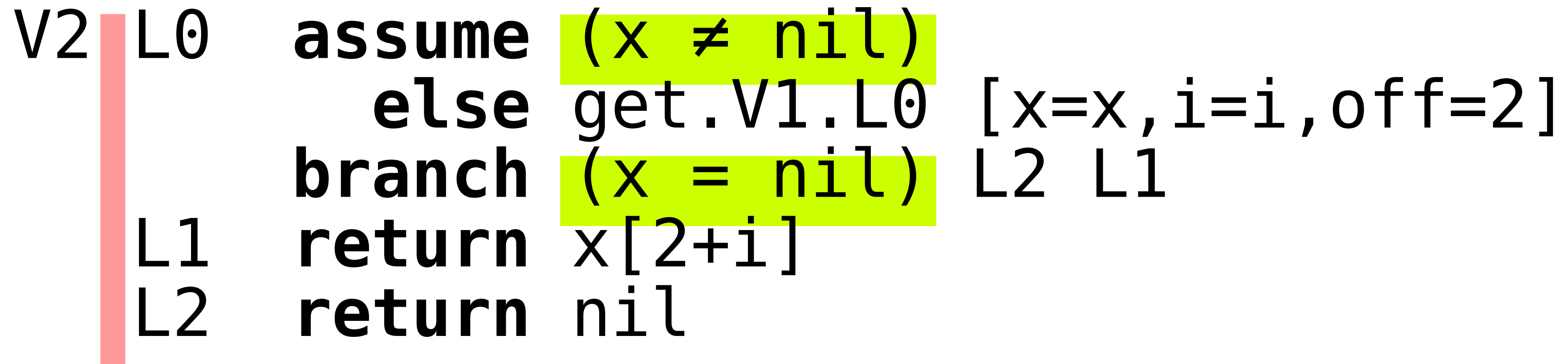


Speculation



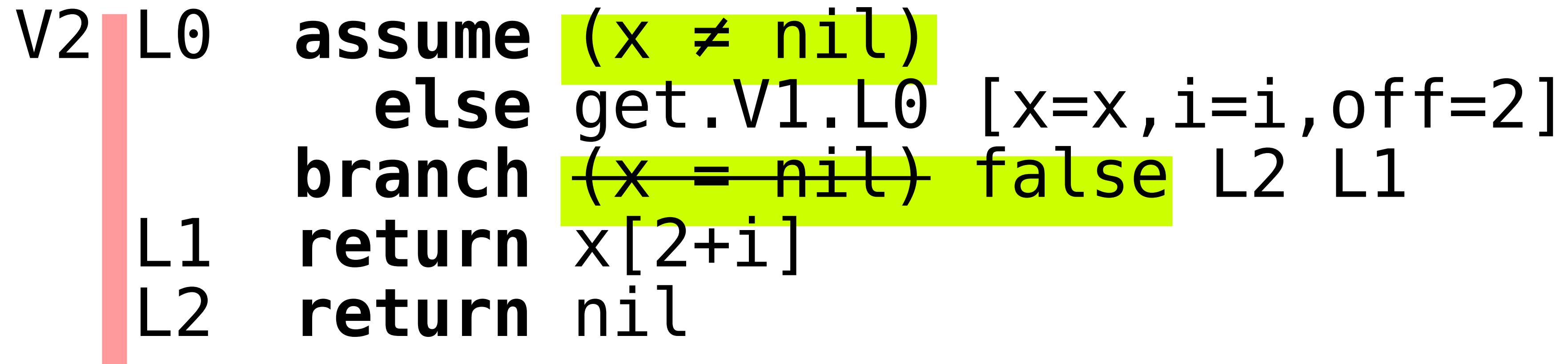
Speculation

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

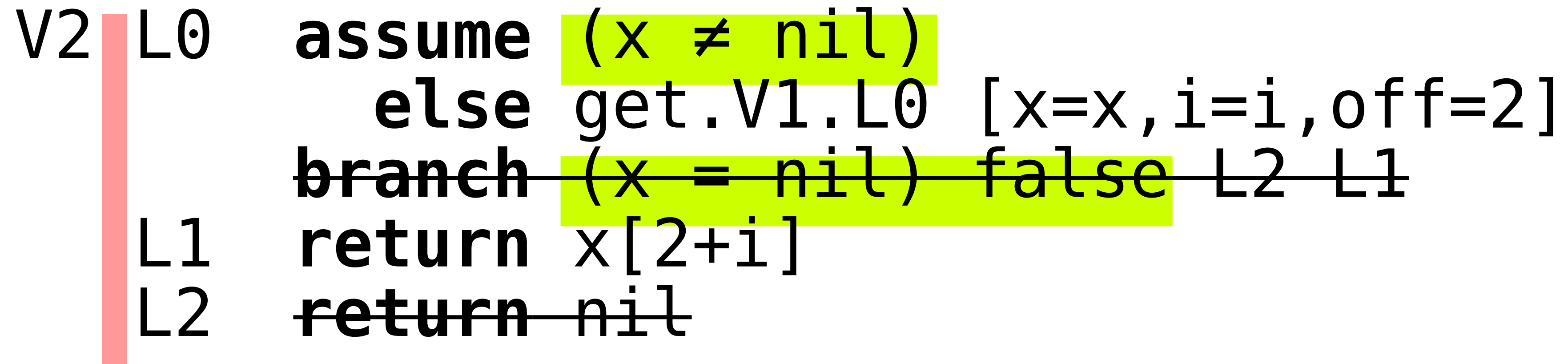


Speculation

get(x, i)



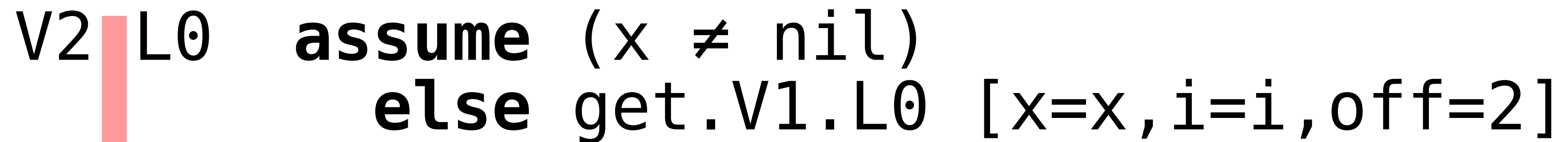
Speculation



Speculation

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

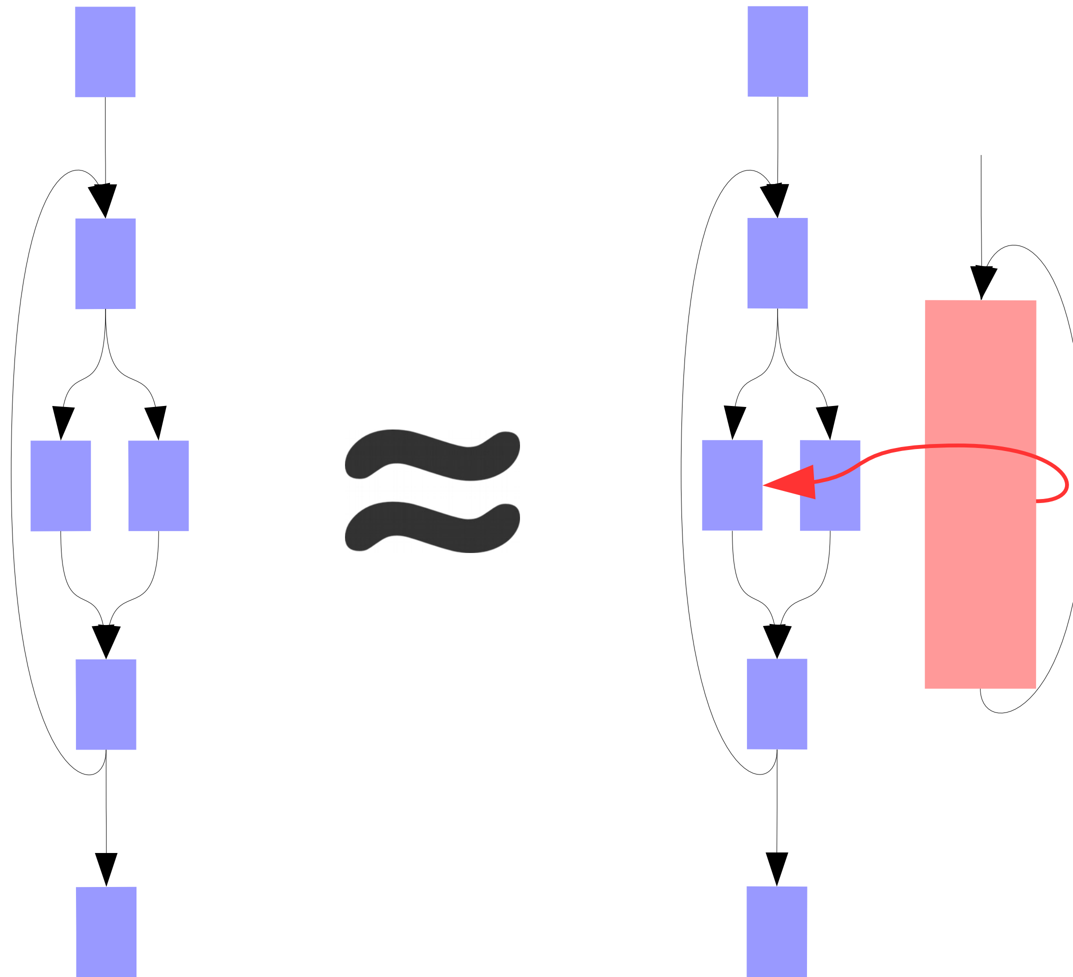
```
 V1      …
```



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

    L1  return x[2+i]
```

Correctness

Proof Structure



Case Study: R

Me: how hard can it be?

l) scopes

```
x <- 1
f <- function(a) x + a
g <- function() {
    x <- 2
    f(0)
}
g() # what does this return?
```

```
x <- 1
f <- function(a) x + a
g <- function() {
    x <- 2
    f(0)
}
g() # what does this return?
```

```
x <- 1
f <- function(a) x + a
g <- function() {
    x <- 2
    f(0)
}
g() # what does this return?
```

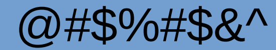
```
x <- 1
f <- function(a) x + a
g <- function() {
      x <- 2
      f(0)
}
g()  #→ 1
```

```
x <- 1
f <- function(a) x + a
g <- function() {
    x <- 2
    f(0)
}
g()  #→ 1
```

```
x <- “gotcha”
g <- function(a) {
    x <- 1
    rm(list=a)
    x
}
g(“x”)
```

```
x <- “gotcha”
g <- function(a) {
    x <- 1
    rm(list=a)
    x
}
g(“x”)
```

```
x <- “gotcha”
g <- function(a) {
      x <- 1
      rm(list=a)
      x
}
g(“x”)  #→  “gotcha”
```



```
x <- “gotcha”
g <- function(a) {
    x <- 1
    rm(list=a)
    x
}
g(“x”)  #→  “gotcha”
```

```
x <- “gotcha”
g <- function(a) {
      x <- 1
      rm(list=a)
      x
}
g(“x”)  #→  “gotcha”
```

II) promises

```
f <- function(x) {
  print(“a”)
  x
}
f(print(“b”))
```

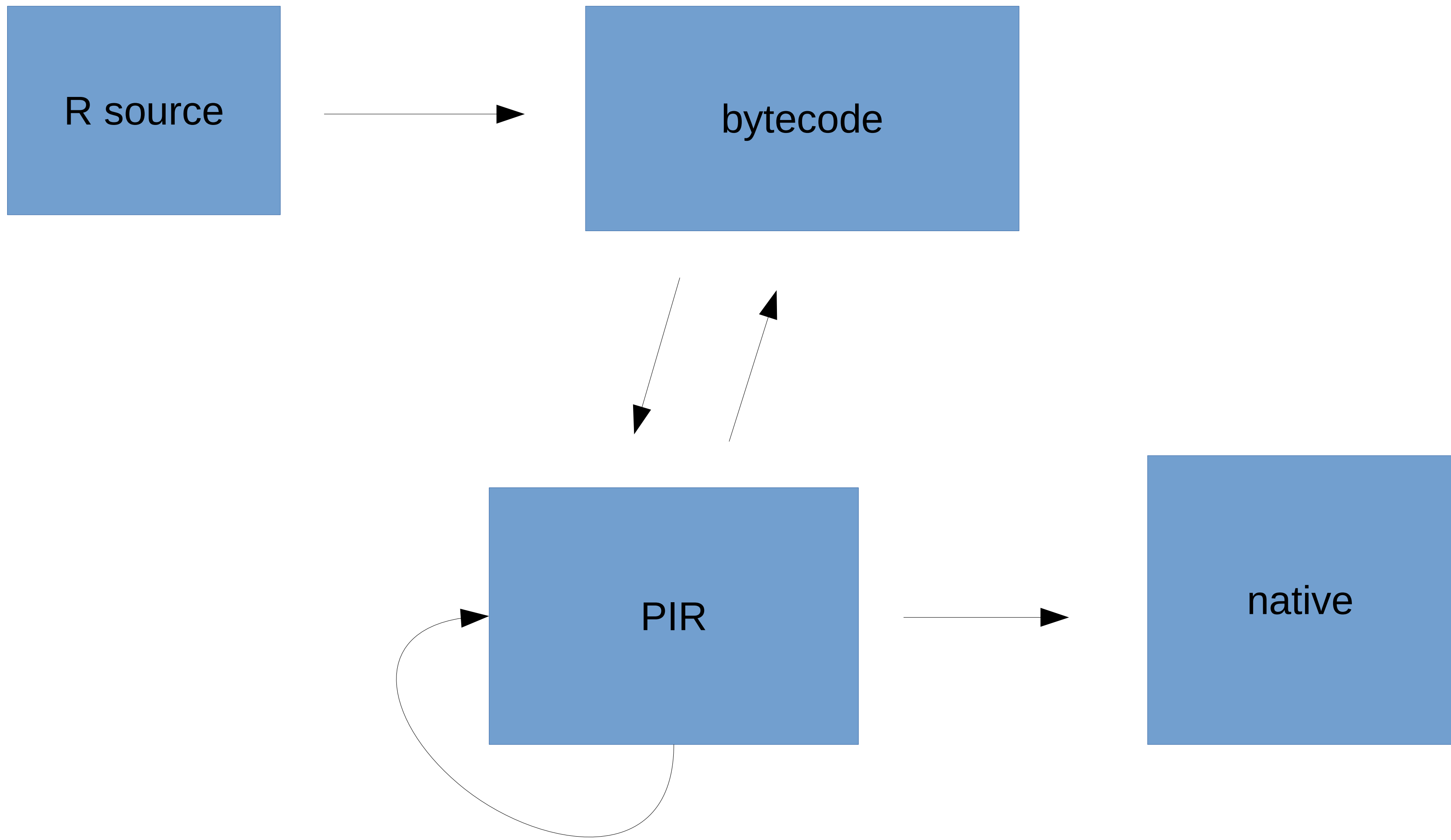
```
f <- function(x) {
  print(“a”)
  x
}
f(print(“b”))  # → “a, b”
```

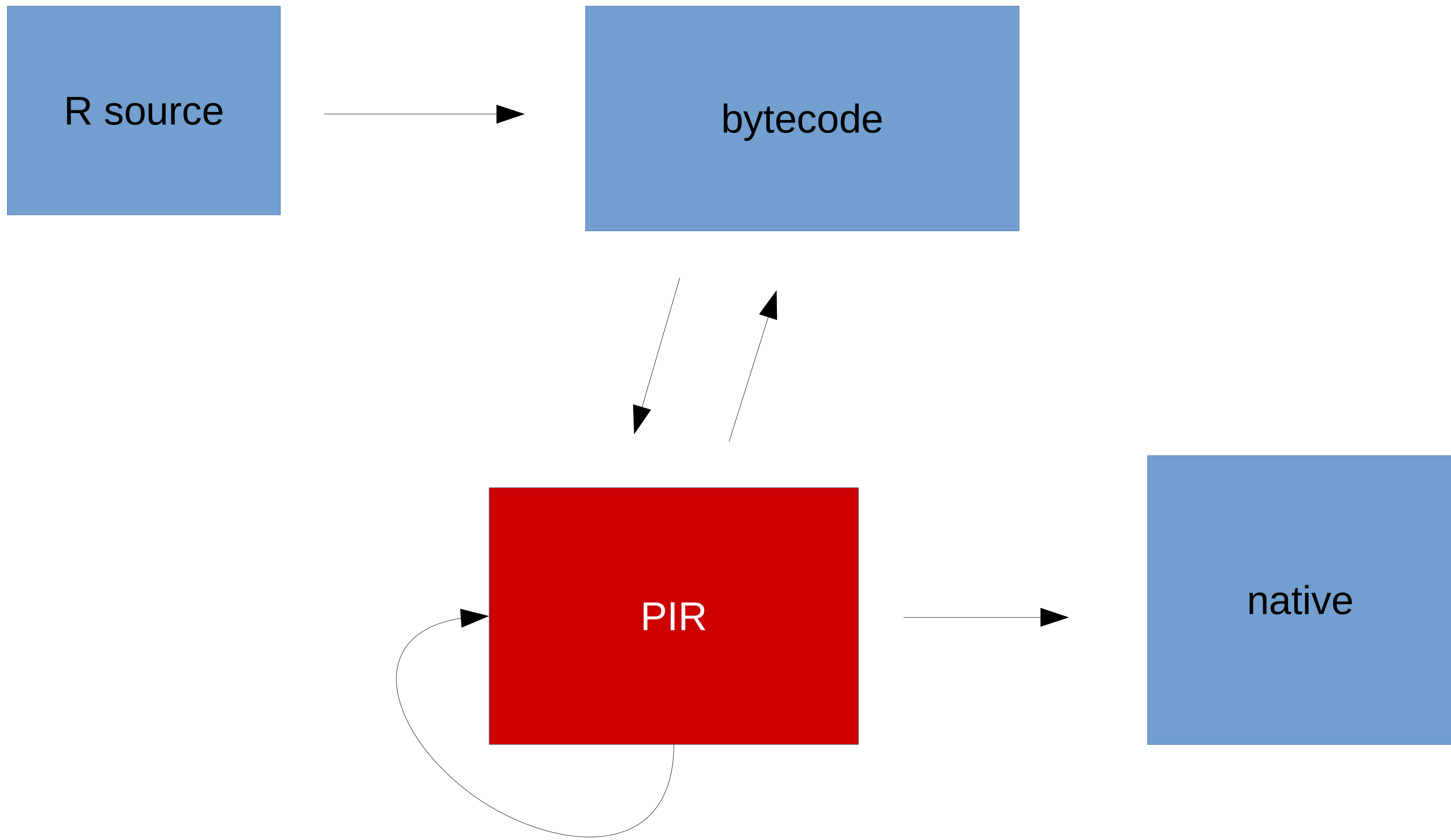
```
f <- function(x) {
  print("a")
  x
}
f(print("b"))  # → "a, b"
```

`print("b")` | env | __result__

- demo

Ř


R source
bytecode
PIR
native


R source
bytecode
PIR
native

modeling first-class scopes and promises

```
a <- 1
f <- function(b) b
f(a)
```

```
a <- 1
f <- function(b) b
f(a)
```

main

%7	=	LdConst [1] 1
%8	=	StVar *a*, %7, G
%9	=	MkClosure (*f*, G)
%10	=	MkArg (*pr*0, G)
%11	=	Call %9 (%10) G
		…

pr0

%12	=	LdVar *a*, G
		Return %12

```
a <- 1
f <- function(b) b
f(a)
```

f

```
%1  = LdArg 0
e2  = MkEnv (b = %1) parent = G
%3  = Force (%1) e2
      Return %3
```

main

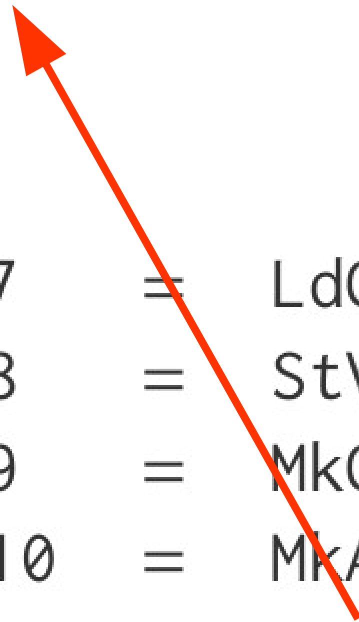
```
%7  = LdConst [1] 1
%8  = StVar a, %7, G
%9  = MkClosure (f, G)
%10 = MkArg (pr0, G)
%11 = Call %9 (%10) G
      ...
```

pr0

```
%12 = LdVar a, G
      Return %12
```

```
a <- 1
f <- function(b) b
f(a)
```

```
f
    %1  =  LdArg 0
    e2  =  MkEnv (b = %1) parent = G
    %3  =  Force (%1) e2
           Return %3
```



```
main
    %7   =  LdConst [1] 1
    %8   =  StVar a, %7, G
    %9   =  MkClosure (f, G)
    %10  =  MkArg (pr0, G)
    %11  =  Call %9 (%10) G
            ...
pr0
    %12  =  LdVar a, G
            Return %12
```

```r
a <- 1
f <- function(b) b
f(a)
```

main

```
%7   =  LdConst [1] 1
%8   =  StVar a, %7, G
%10  =  MkArg (pr0, G)

e2   =  MkEnv (b = %10) parent = G
%3   =  Force (%10) e2
```

pr0

```
%12  =  LdVar a, G
        Return %12
```

```
a <- 1
f <- function(b) b
f(a)
```

```
main
   %7   =  LdConst [1] 1
   %8   =  StVar a, %7, G
   %10  =  MkArg (pr0, G)

   e2   =  MkEnv (b = %10) parent = G
   %3   =  Force (%10) e2


 pr0
   %12  =  LdVar a, G
           Return %12
```

```
a <- 1
f <- function(b) b
f(a)
```

main

```
%7   = LdConst [1] 1
%8   = StVar a, %7, G
%10  = MkArg (pr0, G)

e2   = MkEnv (b = %10) parent = G
%3   = Force (%10) e2
```

pr0

```
%12  = LdVar a, G
       Return %12
```

```
a <- 1
f <- function(b) b
f(a)
```

main

$$
\begin{aligned}
\%7 &= \texttt{LdConst}\ [1]\ 1 \\
\%8 &= \texttt{StVar}\ a,\ \%7,\ G \\
\%10 &= \texttt{MkArg}\ (pr0,\ \texttt{e8}) \\
\texttt{e2} &= \texttt{MkEnv}\ (b = \%10)\ \text{parent} = G \\
\%12 &= \texttt{LdVar}\ a,\ G \\
\%3 &= \texttt{Force}\ (\%12)\ \texttt{e2}
\end{aligned}
$$

```
a <- 1
f <- function(b) b
f(a)
```

main

$\%7 \quad = \quad \texttt{LdConst}\ [1]\ 1$

$\%8 \quad = \quad \texttt{StVar}\ a,\ \%7,\ G$

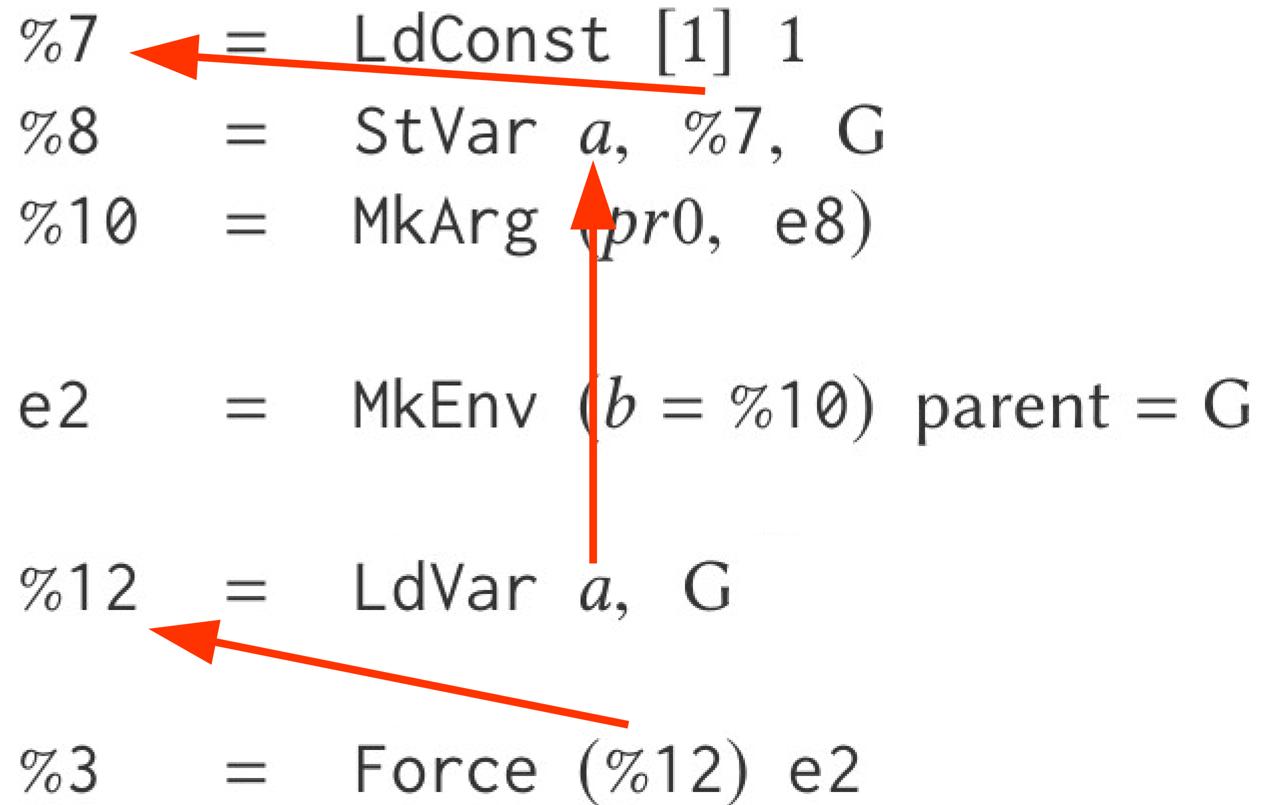
$\%10 \quad = \quad \texttt{MkArg}\ (pr0,\ \texttt{e8})$

$\texttt{e2} \quad = \quad \texttt{MkEnv}\ (b = \%10)\ \text{parent} = G$

$\%12 \quad = \quad \texttt{LdVar}\ a,\ G$

$\%3 \quad = \quad \texttt{Force}\ (\%12)\ \texttt{e2}$

```
a <- 1
f <- function(b) b
f(a)
```



Compiler Correctness

S – source language
T – target language
→ – compiler

S – source language

T – target language

→ – compiler

if $S \rightarrow T$

then $S =_{st} T$

S – source language
T – target language
→ – compiler

if $S \rightarrow T$
then $S \;{}_{s}{=}_{t}\; T$

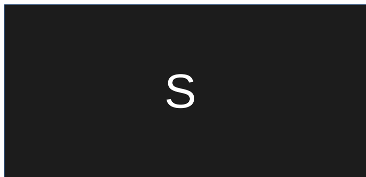
(whole program correctness)

observational equivalence

$$S \; {}_{s}{=}_{t} \; T$$

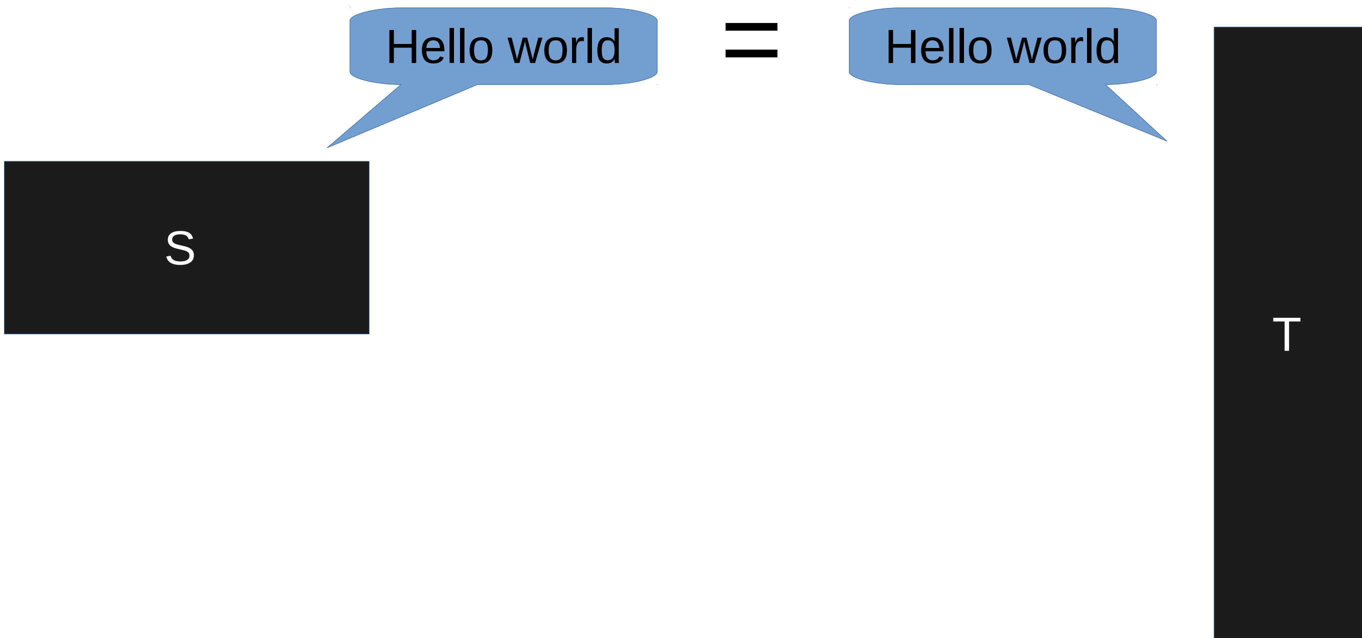
observational equivalence

$$S \; {}_{s}{=}_{t} \; T$$

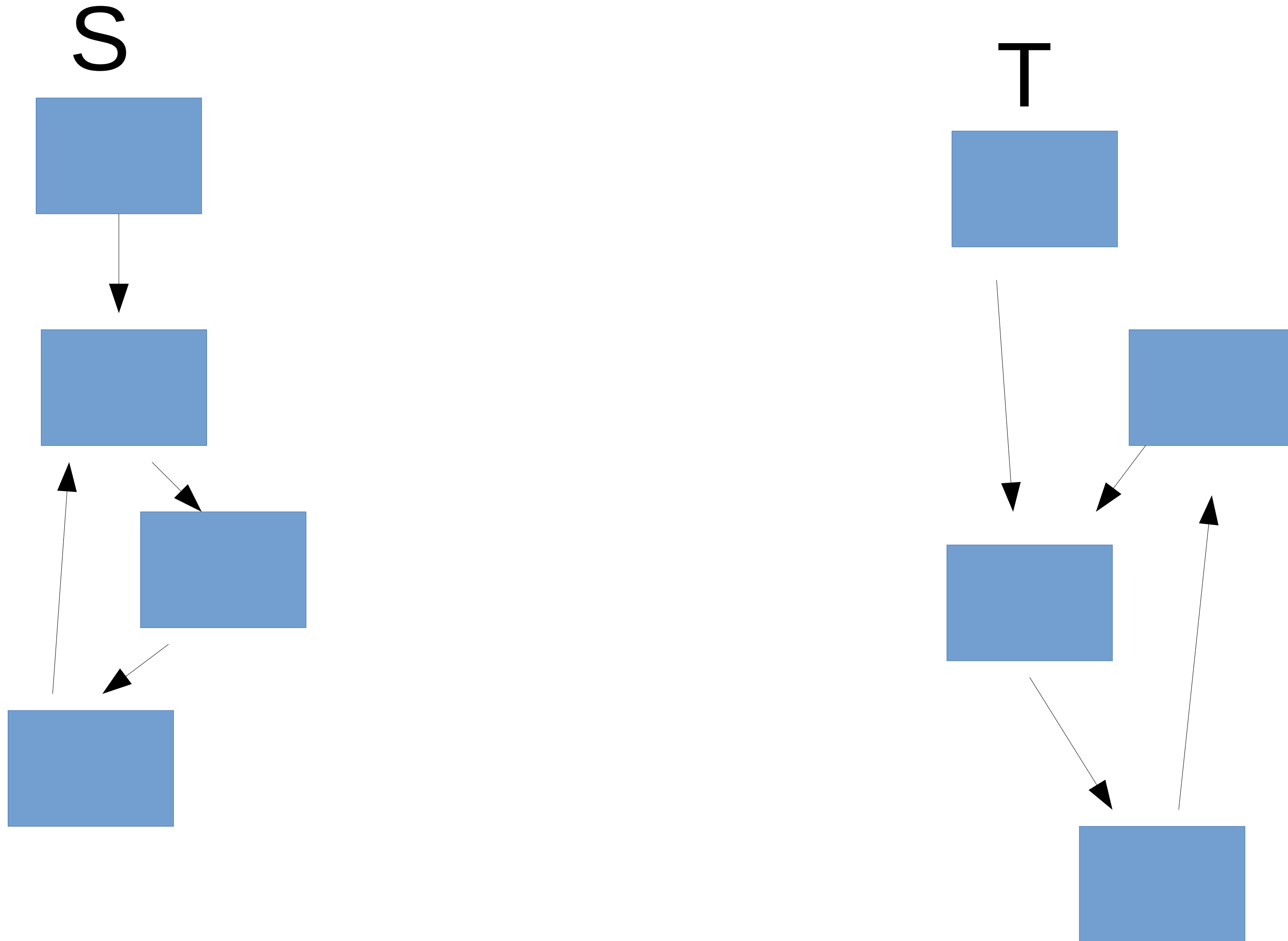


observational equivalence

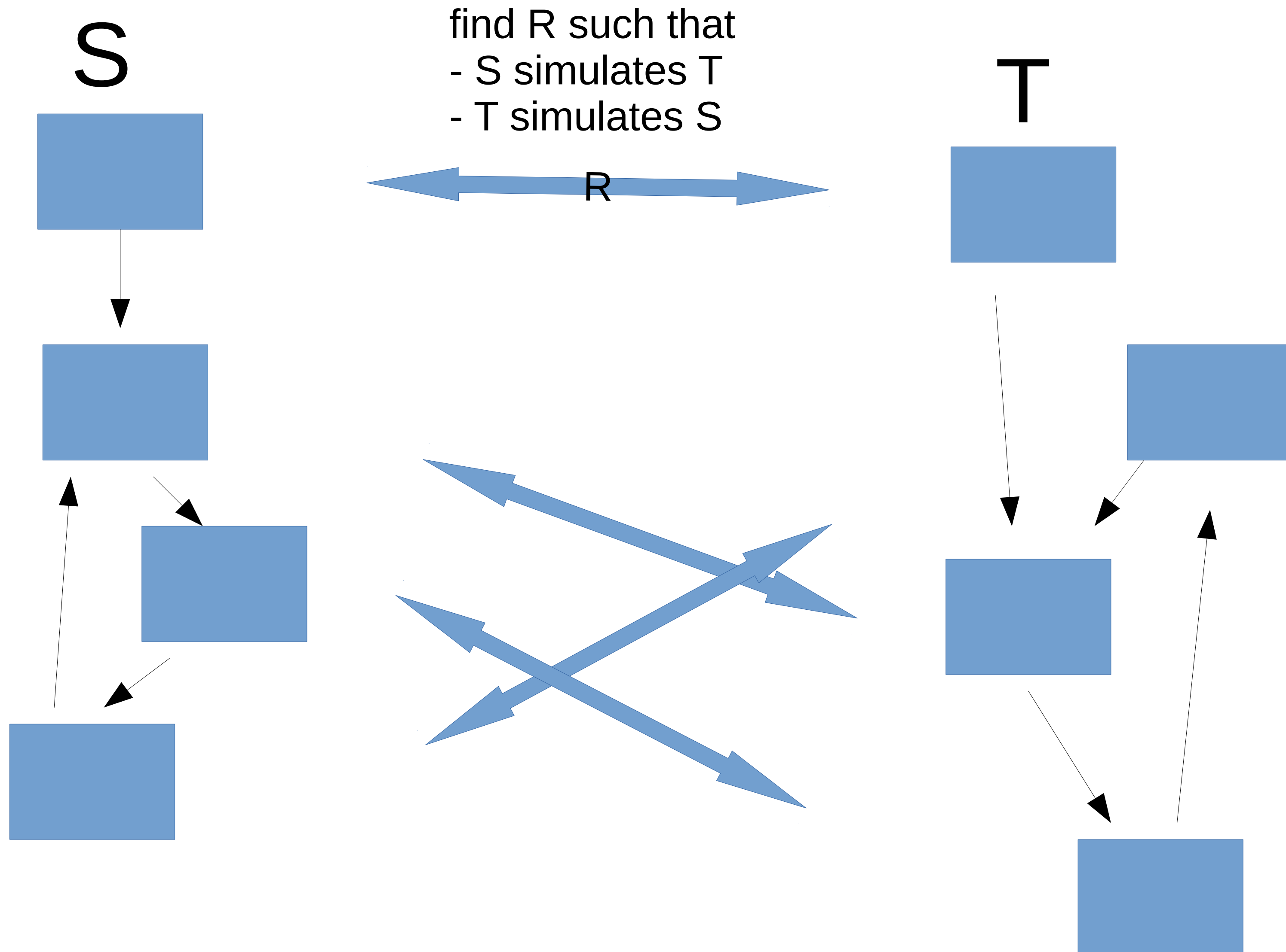
$$S \;{}_{s}{=}_{t}\; T$$



Bi-Simulation



Bi-Simulation



Bi-Simulation