Compilation and Program Analysis (#13) : Beyond ahead-of-time imperative compilation

Gabriel Radanne

Master 1, ENS de Lyon et Dpt Info, Lyon1

November 6, 2024

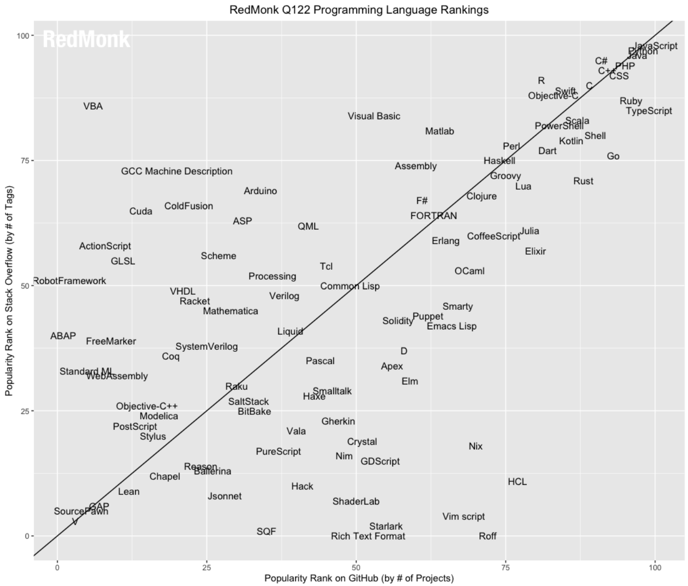
Compilation in this course:

- Start from an imperative core language
- Compile statically to a binary executable
- Classical intermediate representations and algorithms

Compilation in this course:

- Start from an <u>imperative</u> core language
- Compile <u>statically</u> to a binary executable
- Classical intermediate representations and algorithms

$\Rightarrow$ But language design didn't stop at C!


RedMonk Q122 Programming Language Rankings
RedMonk
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75
100
JavaScript
Python
Java
C#
PHP
C++
CSS
R
Swift
C
Objective-C
Ruby
TypeScript
VBA
Visual Basic
Matlab
Scala
PowerShell
Shell
Kotlin
Perl
Dart
Go
Haskell
Assembly
GCC Machine Description
Groovy
Lua
Rust
Arduino
Clojure
F#
ColdFusion
Cuda
FORTRAN
ASP
QML
Julia
CoffeeScript
Erlang
ActionScript
Elixir
Scheme
GLSL
Tcl
Processing
OCaml
RobotFramework
Common Lisp
VHDL
Racket
Verilog
Mathematica
Smarty
Puppet
Solidity
Emacs Lisp
Liquid
ABAP
FreeMarker
SystemVerilog
D
Coq
Pascal
Apex
Standard ML
WebAssembly
Elm
Raku
Smalltalk
Haxe
SaltStack
Objective-C++
BitBake
Modelica
Gherkin
PostScript
Vala
Stylus
Crystal
Nix
PureScript
Nim
GDScript
Reason
Ballerina
Chapel
HCL
Hack
Lean
Jsonnet
ShaderLab
GAP
SourcePawn
Vim script
V
Starlark
SQF
Rich Text Format
Roff

What strategy should we use to approach:

- parallel and concurrent features (in C/C++, Java, . . .)
- dynamic languages (Javascript, Ruby, Python, Scheme, . . .)
- hostile languages (PHP, Perl, R, Javascript, Ruby, . . .)
- objects (Java, Javascript, C++, C#, . . .)
- Data manipulation (SQL, GraphQL, Python for datascience, . . .)
- weird features (OCaml, R, Haskell, Rust)

Enough to fill many courses!

Sources of inspiration used for these slides

The SSA book (Chapter 6 by Lennart Beringer)

SSA is functional programming (Andrew Appel — 1998)

A Correspondence between Continuation Passing Style (Richard Kelsey — 1995)

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```

In normal CFG:

In SSA form:

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```

In normal CFG:

- the variable `x` being assigned to is carefully distinguished from the expression to the right

In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```

In normal CFG:

- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time

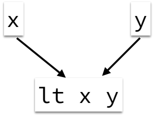
In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
…
a <- lt x y
```



In normal CFG:

- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time
- use sites can map to various def-sites

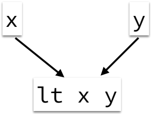
In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value
- use sites can be thought of as data flow graph edges

Looking through the SSA glass

Consider a simple couple of instructions:

```
x <- add y z
...
a <- lt x y
```



<u>In normal CFG</u>:

- the variable `x` being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time
- use sites can map to various def-sites

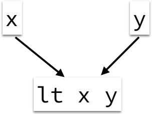
<u>In SSA form</u>:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value
- use sites can be thought of as data flow graph edges

Looking through the SSA glass

Consider a simple couple of instructions:

```
_ <- add y z
...
a <- lt _ y
```



In normal CFG:

- the variable x being assigned to is carefully distinguished from the expression to the right
- instructions are computations to be processed:
  the meaning of their evaluation can be compromised at any time
- use sites can map to various def-sites

In SSA form:

- names are globally unique. So assignments and def-sites of variables are in bijection
- the instruction itself can be thought of as a static value
- use sites can be thought of as data flow graph edges

Uses of variables can be represented as simple pointers to their defining instructions
(and LLVM do so to represent programs in memory!)

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions `fun f(x,y) = e`
- let-binding constructs `let x = e1 in e2`

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let x = e1 in e2
```

e2 is the static scope
of variable x

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea:
static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions

```
fun f(x,y) = e
```

- let-binding constructs

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea:
static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions

```
fun f(x,y) = e
```

- let-binding constructs

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions
- let-binding constructs

```
fun f(x,y) = e
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Binding in the functional world

The idea and application of the SSA form stems from the imperative world.
But in the mean time, the functional world has been doing their own compilation!

The job is of course seemingly quite distinct. We are now fundamentally playing with:

- (Mutually recursive) functions

```
fun f(x,y) = e
```

- let-binding constructs

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

And when it comes to the central purpose of all this story, variables, we rely on a powerful idea: static scopes!

Unicity of names is unnecessary, but can be enforced by alpha renaming

Binding in the functional world

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

Binding in the functional world

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Binding in the functional world

Nearest enclosing binding

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Binding in the functional world

Nearest enclosing binding

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

<u>Binding shadows</u>: enforces that each use-site maps to a unique def-site

Binding in the functional world

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Binding in the functional world

```
fun f(z) =
    let v = 3 in
      let y = (let v = 2 * v in 4 * v)
      in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Binding in the functional world

```
fun f(z) =
    let v = 3 in
      let y = (let v = 2 * v in 4 * v)
      in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Each use of a variable is dominated by its unique definition!

Binding in the functional world

```
fun f(z) =
    let v = 3 in
      let y = (let v = 2 * v in 4 * v)
      in y * v + z
```

Binding shadows: enforces that each use-site maps to a unique def-site

Well scoped: the only fresh variables are formal arguments

Each use of a variable is dominated by its unique definition!

Uniqueness by scope and not by name: referential transparency!

Referential transparency: compositional equational reasoning!

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in k(y * v + z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
let k = λx . 2 * x in
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in k(y * v + z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
fun f(k) =
```

```
let v = 3 in
  let y = (let v = 2 * v in 4 * v)
  in k(y * v + z)
```

```
in let k = λx . 2 * x in f(k)
```

Control flow in the functional world

```
fun f(y,k) =
  let x = 4 in
  let k’ = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k’(z)
    else let z = 3 in k’(z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k’ = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k’(z)
    else let z = 3 in k’(z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Craft a new local one

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Craft a new local one

Share its use between two branches

Control flow in the functional world

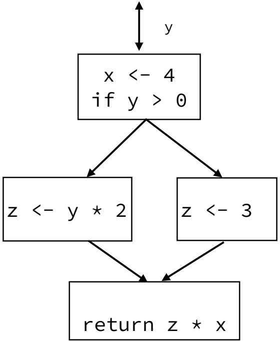
CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

parameterized by a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```

Craft a new local one

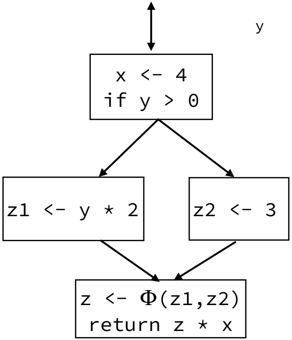
Share its use between two branches



Control flow in the functional world

<u>CPS style</u>: where an imperative compiler would carry on a return address, the functional one calls a continuation

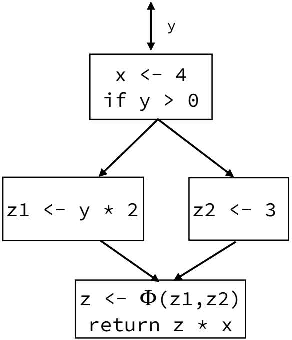
```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k'(z)
    else let z = 3 in k'(z)
```



Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```



continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

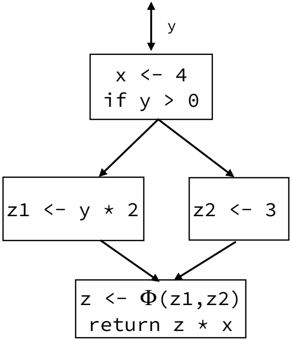
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

z <- Φ(z1,z2)

```
fun f(y,k) =
  let x = 4 in
  let k’ = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k’(z1)
    else let z2 = 3 in k’(z2)
```



continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

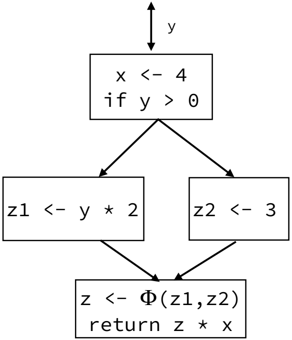
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address, the functional one calls a continuation

```
z <- Φ(z1,z2)
```

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```



continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

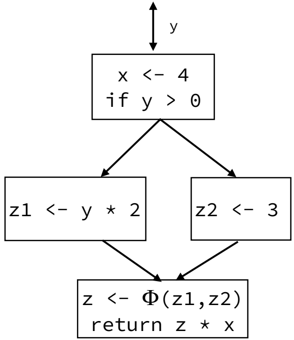
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

<u>CPS style</u>: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
z <- Φ(z1,z2)
```

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```



continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

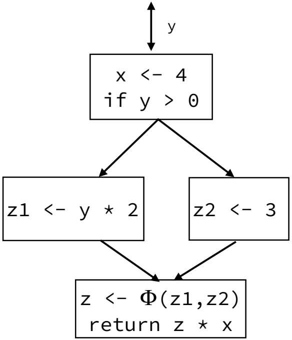
scope of a let-bind <-> dominance region of an assignment

Control flow in the functional world

CPS style: where an imperative compiler would carry on a return address,
the functional one calls a continuation

```
z <- Φ(z1,z2)
```

```
fun f(y,k) =
  let x = 4 in
  let k' = λz. k(z*x)
  if y > 0
    then let z1 = y * 2 in k'(z1)
    else let z2 = 3 in k'(z2)
```

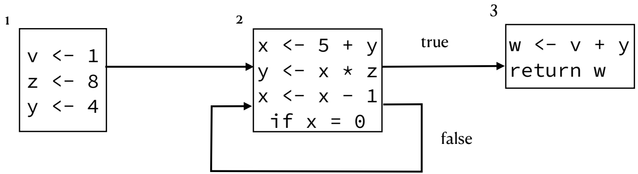


continuation <-> phi-node

calls to the continuation <-> arguments to the phi-node

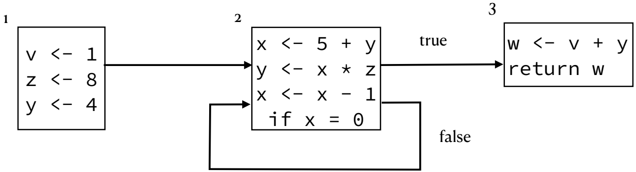
scope of a let-bind <-> dominance region of an assignment

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

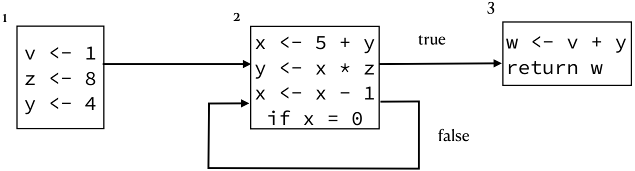
CPS-style

```
fun f(y,k) =
  let x = 4 in
  let k’ = λz. k(z*x)
  if y > 0
    then let z = y * 2 in k’(z)
    else let z = 3 in k’(z)
```

(let-normal) direct-style (i.e. with tail recursive calls)

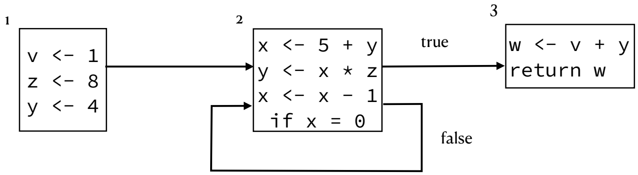
```
fun f(y) =
  let x = 4 in
  fun g(z) = z*x
  if y > 0
    then fun h1() = let z = y * 2 in g(z)
         in h1()
    else fun h2() = let z = 3 in g(z)
         in h2()
```

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

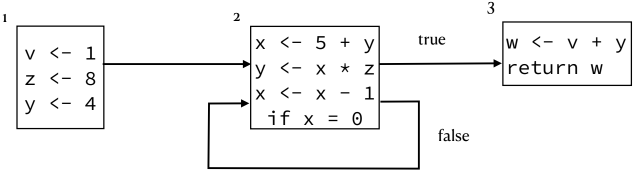
SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

SSA: functional construction

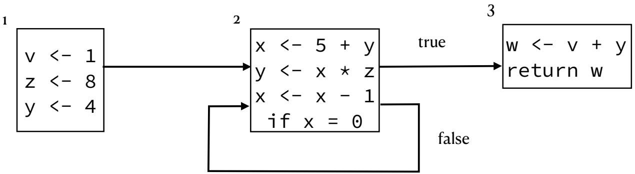


We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

```
fun f1()        = let v = 1, z = 8, y = 4
                  in f2(v,z,y)
```

SSA: functional construction

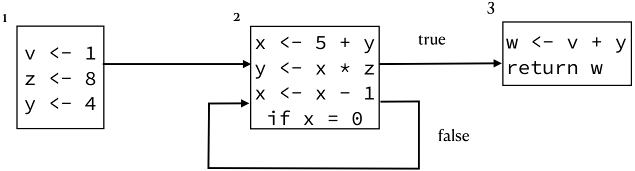


We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

```
fun f1()        = let v = 1, z = 8, y = 4
                  in f2(v,z,y)
fun f2(v,z,y) = let x = 5 + y, y = x * z, x = x - 1
                  in if x = 0
                     then f3(y,v)
                     else f2(v,z,y)
```

SSA: functional construction



We are going to turn the CFG above in functional style, but via its functional representation

Liveness analysis + one mutually recursive function per block

```
fun f1()       = let v = 1, z = 8, y = 4
                 in f2(v,z,y)
fun f2(v,z,y) = let x = 5 + y, y = x * z, x = x - 1
                 in if x = 0
                    then f3(y,v)
                    else f2(v,z,y)
fun f3(y,v)    = let w = v + y
                 in w
```

SSA: functional construction

- All functions declarations are closed
- Unique definition-site per use is satisfied
- In a let binding `let x = e1 in e2`, the subterm `e2` corresponds to the successor in the control flow of the assignment to `x`

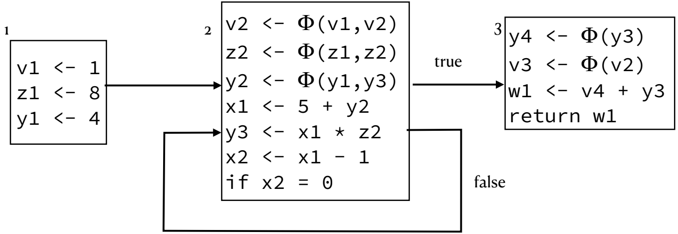
```
fun f1()      = let v = 1, z = 8, y = 4
                in f2(v,z,y)
fun f2(v,z,y) = let x = 5 + y, y = x * z, x = x - 1
                in if x = 0
                   then f3(y,v)
                   else f2(v,z,y)
fun f3(y,v)   = let w = v + y
                in w
```

SSA: functional construction

- All functions declarations are closed
- Unique definition-site per use is satisfied
- In a let binding `let x = e1 in e2`, the subterm `e2` corresponds to the successor in the control flow of the assignment to `x`

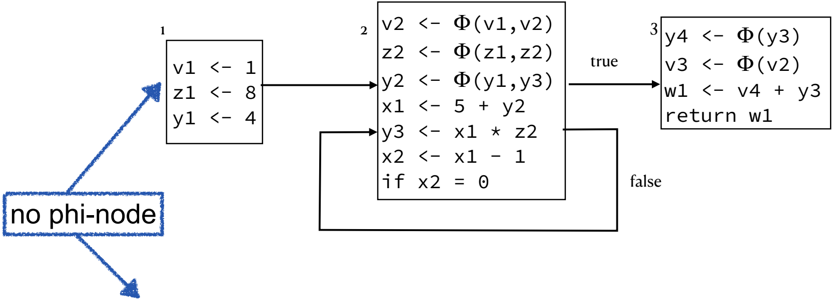
```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

SSA: functional construction



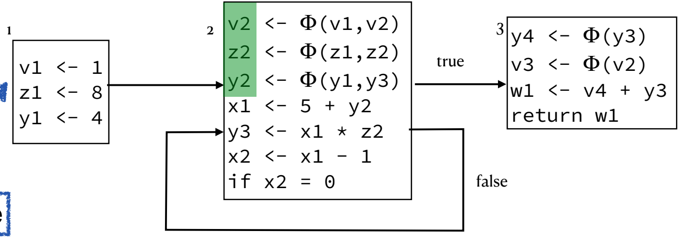

```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

SSA: functional construction




```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

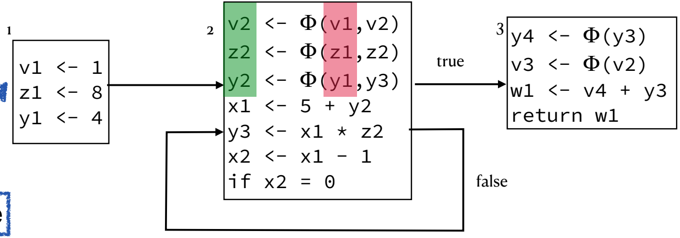
SSA: functional construction



no phi-node

```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2)   = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

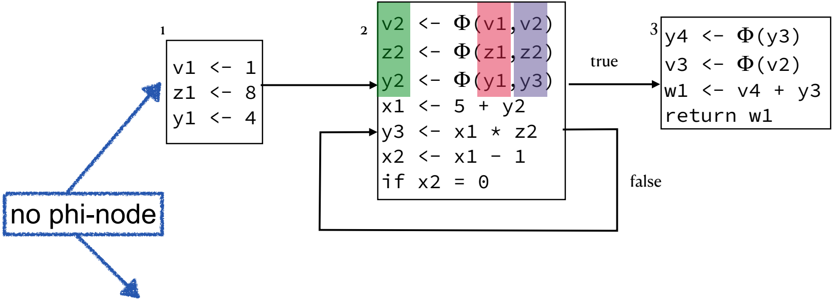
SSA: functional construction



no phi-node

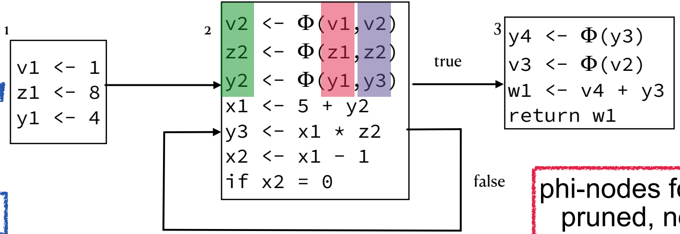
```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2)  = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

SSA: functional construction




```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2)   = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

SSA: functional construction



no phi-node

phi-nodes for all live variable:
pruned, non-minimal SSA

```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2)   = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

Block sinking

```
fun f1()          = let v1 = 1, z1 = 8, y1 = 4
                    in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                    in if x2 = 0
                       then f3(y3,v2)
                       else f2(v2,z2,y3)
fun f3(y4,v3)     = let w1 = v4 + y3
                    in w1
```

Block sinking

```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2) = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

Block sinking

```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2)   = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

Block sinking

```
fun f1()           = let v1 = 1, z1 = 8, y1 = 4
                     in f2(v1,z1,y1)
fun f2(v2,z2,y2)   = let x1 = 5 + y2, y3 = x1 * z2, x2 = x1 - 1
                     in if x2 = 0
                        then f3(y3,v2)
                        else f2(v2,z2,y3)
fun f3(y4,v3)      = let w1 = v4 + y3
                     in w1
```

Essentially: based on the DT of the call graph
The dominance relationship becomes apparent in the scoping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3(y,v) = let w = y + v in w
              in f3(y,v)
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(v,z,y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(v,z,y)
   in f2(v,z,y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

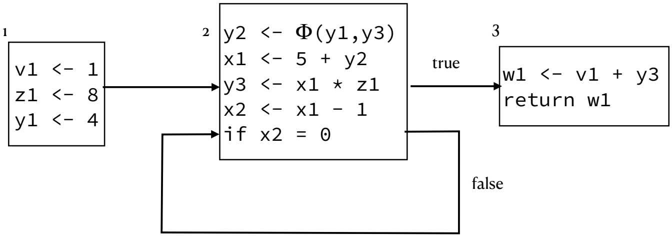
Parameter dropping

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(y)
   in f2(y)
in f1()
```

We drop the formal parameters that can be statically ruled out as semantically irrelevant

Minimal SSA form

```
fun f1() =
   let v = 1, z = 8, y = 4
   in fun f2(y) =
      let x = 5 + y, y = x * z, x = x - 1
      in if x = 0
         then fun f3() = let w = y + v in w
              in f3()
         else f2(y)
   in f2(y)
in f1()
```



Summary – SSA and CPS

The Church-Turing Hypothesis of IR !

- Single Static Assignment: quintessentially **imperative**
  Good at intra-procedural (constant propagation, dead code, allocation, ...)
- Continuation Passing Style: quintessentially **functional**
  Good at inter-procedural (inlining!)

They are equivalent!

Highlight the importance of information sharing.

$\Rightarrow$ Compilers can use either for optimisations, and go from one to the other.

Introduction: Algebraic Data Types

Product types (a.k.a. records, tuples, structs)

```
type point = { x: int, y: int }
let distance p1 p2 =
 let dx = p2.x -p1.x in
 let dy = p2.y -p1.y in
 sqrt(dx * dx + dy * dy)
```

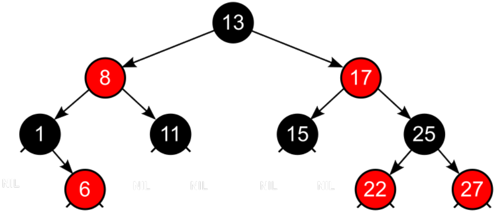
Sum types (a.k.a. enums, tagged unions)

```
type Card = King | Queen | Jack | @Numeral of int@
let value c = match c with
  | King -> 13
  | Queen -> 12
  | Jack -> 11
  | Numeral(n) -> n
```

Introduction: Algebraic Data Types

Example of red-black trees

```
type color = Red | Black
type rbt =
| Empty
| Node of Color * int * RBT * RBT
```




```
let cardinal (t: RBT) : int = match t with
  | Empty -> 0
  | Node(_, _, t1, t2) -> 1 + cardinal(t1) + cardinal(t2)
```

Introduction: Algebraic Data Types

Example of red-black trees

```
type color = Red | Black
type rbt =
| Empty
| Node of Color * int * RBT * RBT
```

Rebalancing operation:

```
match c, v, t1, t2 {
| Black, z, Node(Red, y, Node(Red, x, a, b), c), d
| Black, z, Node(Red, x, a, Node(Red, y, b, c)), d
| Black, x, a, Node(Red, z, Node(Red, y, b, c), d)
| Black, x, a, Node(Red, y, b, Node(Red, z, c, d))
  -> Node(Red, y, Node(Black, x, a, b), Node(Black, z, c, d))
| a, b, c, d -> Node (a, b, c, d)
```

- Type safety, exhaustivity and non-redundancy checks
- Complex nested patterns are expressive yet concise

- How to execute this?
- How to compile this?

Let's try in practice

Try to compile this code to if-tests:

```
let f x y z = match x,y,z with
  | _, false, true -> 1
  | false, true, _ -> 2
  | _, _, false -> 3
  | _, _, true -> 4
```

A more complex example

$$\tau_0 = \textit{None} + \textit{Some}(A + B + C(u32))$$

```
match v with
| (None | Some(A)) -> 0
| Some(B) -> 1
| Some(C(n)) -> 2 + n
```

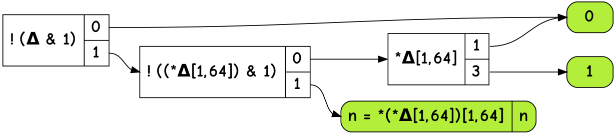
Let’s try it.

OCaml representation-specific pattern matching

Input

```
match v with
| (None | Some(A)) -> 0
| Some(B) -> 1
| Some(C(n)) -> 2 + n
```

Output decision tree (as graph)



Output decision tree (as C code)

```
switch(v & 1) {
 case 0: // Some (pointer to ...)
  switch((*v)[1] & 1) {
   case 0: // C
    uint32_t n = (*((*v)[1]))[1] >> 1;
    return 2 + n;
   case 1: // unit variant (A or B)
    switch((*v)[1]) {
     case 0b01: // A
      return 0;
     case 0b11: // B
      return 1;}}
 case 1: // None: last bit is 1 (non ptr)
  return 0;}
```

An Intermediate Representation!

- We have a pattern matching problem
- We want a decision tree

$\Rightarrow$ We need an appropriate intermediate representation!

The pattern Matrix

Let's try to represent a matching problem in its globality

```
match v with
| (None | Some(A)) -> 0
| Some(B) -> 1
| Some(C(n)) -> 2 + n
```

$$\Rightarrow \left(\begin{array}{c|c} . & \\ \hline \textit{None} \mid \textit{Some}(A) & \emptyset, 0 \\ \textit{Some}(B) & \emptyset, 1 \\ \textit{Some}(C(n)) & \emptyset, 2 \end{array}\right)$$

```
match x,y,z with
| _, false, true -> 1
| false, true, _ -> 2
| _, _, false -> 3
| _, _, true -> 4
```

$$\Rightarrow \left(\begin{array}{ccc|c} .0 & .1 & .2 & \\ \hline \_ & \textit{False} & \textit{True} & \emptyset, 1 \\ \textit{False} & \textit{True} & \_ & \emptyset, 2 \\ \_ & \_ & \textit{False} & \emptyset, 3 \\ \_ & \_ & \textit{True} & \emptyset, 4 \end{array}\right)$$

In the pattern matrix, columns represent "positions" in the input, and lines are patterns and outputs

Compilation scheme

$$\left(\begin{array}{c|c} \cdot & \\ \hline \textit{None} \mid \textit{Some}(A) & \emptyset, 0 \\ \textit{Some}(B) & \emptyset, 1 \\ \textit{Some}(C(n)) & \emptyset, 2 \end{array}\right)$$

- Initial pattern matrix matches the main discriminant against toplevel patterns
- Each case yields its index and an empty binding environment

Compilation scheme

$$\left(\begin{array}{c|c} . & \\ \hline \textit{None} \mid \textit{Some}(A) & \emptyset, 0 \\ \textit{Some}(B) & \emptyset, 1 \\ \textit{Some}(C(n)) & \emptyset, 2 \end{array}\right) \xrightarrow{\text{or}} \left(\begin{array}{c|c} . & \\ \hline \textit{None} & \emptyset, 0 \\ \textit{Some}(A) & \emptyset, 0 \\ \textit{Some}(B) & \emptyset, 1 \\ \textit{Some}(C(n)) & \emptyset, 2 \end{array}\right)$$

Split or-patterns

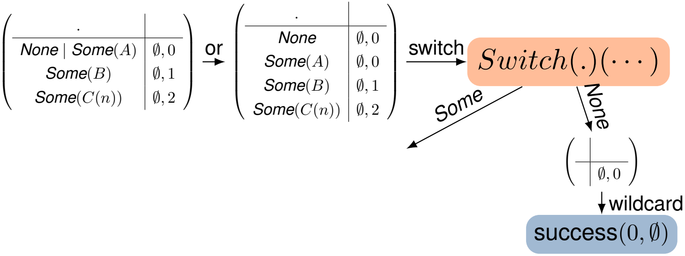
Compilation scheme

$$\left(\begin{array}{c|c} \cdot & \\ \hline \textit{None} \mid \textit{Some}(A) & \emptyset, 0 \\ \textit{Some}(B) & \emptyset, 1 \\ \textit{Some}(C(n)) & \emptyset, 2 \end{array}\right) \xrightarrow{\text{or}} \left(\begin{array}{c|c} \cdot & \\ \hline \textit{None} & \emptyset, 0 \\ \textit{Some}(A) & \emptyset, 0 \\ \textit{Some}(B) & \emptyset, 1 \\ \textit{Some}(C(n)) & \emptyset, 2 \end{array}\right) \xrightarrow{\text{switch}} \boxed{Switch(.)(\cdots)}$$

$$\boxed{Switch(.)(\cdots)} \xrightarrow{\text{Some}} \qquad \boxed{Switch(.)(\cdots)} \xrightarrow{\text{None}}$$

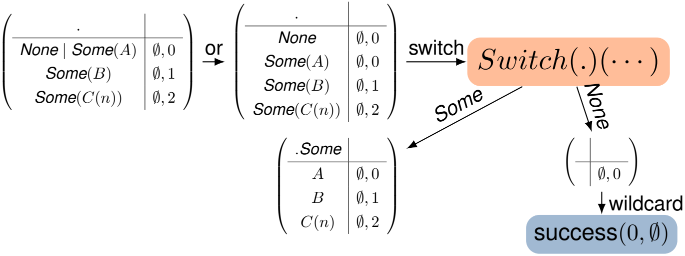
- Retrieve the head constructor of the current subterm, then branch to its associated subtree
- One branch per constructor

Compilation scheme



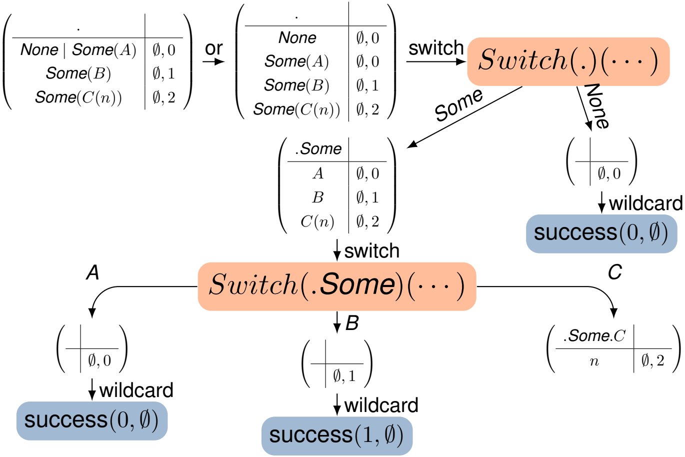
First case accepts any input

Compilation scheme



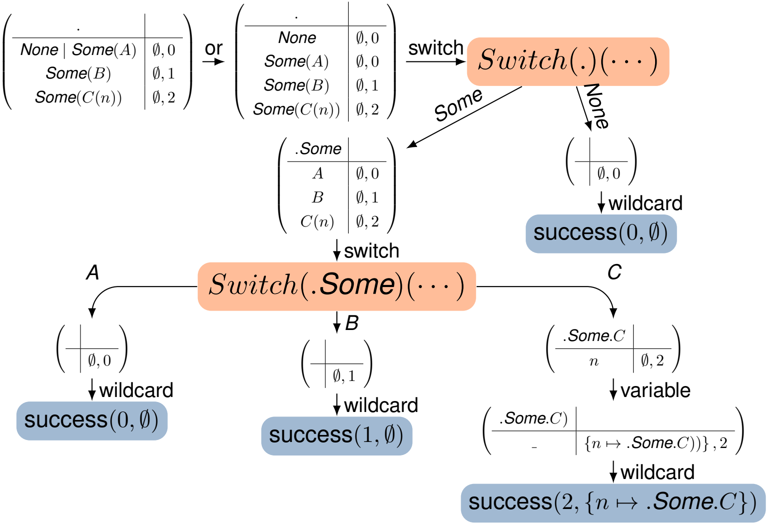
- Discard head-constructor-incompatible cases
- Focus remaining cases on the child subterm

Compilation scheme



Inspect the current subterm

Compilation scheme



- Bind variable pattern to the current subterm

Compilation algorithm – Fail and Success

$\textsc{Compile}\,(\emptyset) = \text{unreachable}$ (No pattern case)

$$\textsc{Compile}\left(\begin{array}{ccc|c} h_1 & \dots & h_n & \\ \hline \boxed{\_ \quad \dots \quad \_} & & & j^1, s^1 \\ \vdots & \ddots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right) = \text{success}(j^1, s^1)$$ (Wildcard case)

Compilation algorithm – Variables

$$\textsc{Compile}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & p_i^1 & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & \cdots\boxed{x}\cdots & p_n^\ell & j^\ell, s^\ell \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & p_i^m & p_n^m & j^m, s^m \end{array}\right) = \textsc{Compile}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & \dots & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & \cdots\boxed{\_}\cdots & p_n^\ell & j^\ell, \boxed{s^\ell \cup s_x} \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right)$$

where $s_x = \{x \to h_i\}$

(Variable case)

Compilation algorithm – Or

$$\textsc{Compile}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & \dots & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & \cdots \boxed{(p \mid q)} \cdots & p_n^\ell & j^\ell, s^\ell \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right) = \textsc{Compile}\left(\begin{array}{ccc|c} h_1 & h_i & h_n & \\ \hline p_1^1 & \dots & p_n^1 & j^1, s^1 \\ \vdots & \vdots & \vdots & \vdots \\ p_1^\ell & \cdots \boxed{p} \cdots & p_n^\ell & j^\ell, s^\ell \\ p_1^\ell & \cdots \boxed{q} \cdots & p_n^\ell & j^\ell, s^\ell \\ \vdots & \vdots & \vdots & \vdots \\ p_1^m & \dots & p_n^m & j^m, s^m \end{array}\right)$$

(Or case)

Compilation algorithm – Switch

$$\textsc{Compile}\left(\mathcal{P}\right) = \begin{cases} i \leftarrow \textsc{PickColumn}(\mathcal{P}) \\ \mathcal{P} = \left(\begin{array}{ccccc|c} h_1 & & h_i & & h_n & \\ \hline p_1^1 & & p_i^1 & & p_n^1 & j^1, s^1 \\ \vdots & & \vdots & & \vdots & \vdots \\ p_1^\ell & \cdots & p_i^\ell & \cdots & p_n^\ell & j^\ell, s^\ell \\ \vdots & & \vdots & & \vdots & \vdots \\ p_1^m & & p_i^m & & p_n^m & j^m, s^m \end{array}\right) \\ \mathit{Tags} = \textsc{GetTags}(p_i^1, \ldots, p_i^m) \\ \forall \mathit{tag} \in \mathit{Tags}, \mathcal{P}_{\mathit{tag}} = \textsc{Expand}(\mathcal{P}, type(h_i), h_i) \\ Switch(h_i)\left\{\mathit{tag} \mapsto \textsc{Compile}\left(\mathcal{P}_{\mathit{tag}}\right)\right\} \end{cases} \qquad \text{(Switch case)}$$

Compilation algorithm – Expand

Inputs		**Outputs**	
τ	*tag*	New Headers	Matrix transformation
$\langle \tau_0, \dots, \tau_l \rangle$		$\begin{pmatrix} h.0 & \cdots & h.\ell \end{pmatrix}$	$\langle p_0, \dots, p_\ell \rangle \mapsto \begin{pmatrix} p_0 & \cdots & p_\ell \end{pmatrix}$ $\_ \mapsto \begin{pmatrix} \_ & \dots & \_ \end{pmatrix}$
$\sum\limits_{1 \leqslant i \leqslant \ell} K_i(\tau_i)$	K_{i_0}	$\begin{pmatrix} h.K_{i_0} \end{pmatrix}$	$K_{i_0}(p) \mapsto \begin{pmatrix} p \end{pmatrix}$ $K_i(\dots) \mapsto \emptyset$ $\_ \mapsto \begin{pmatrix} \_ & \dots & \_ \end{pmatrix}$

Compilation algorithm – Picking a column

How to pick a column?

No clear answer, we want to minimize:

(1) The longest path length, (2) The size of the decision trees.

⇒ Heuristics!

Example of heuristics:

- First row that has a pattern
- Small arity
- The most “needed” columns
- ...

To decision trees ? in OCaml

How do we check the head constructors in reality?

To decision trees ? in OCaml

How do we check the head constructors in reality? $\Rightarrow$ Depends on the language!

Memory Values in OCaml

Blocks

$\boxed{\&_{64}} \longrightarrow \boxed{\text{header}}\boxed{\text{field } 0}\boxed{\cdots}\boxed{\text{field } k-1}$

Unboxed constants

$\boxed{w.1}$ (where $w.1 = (w << 1) \mid 1_{64}$)

$\tau_0 = \mathit{None} + \mathit{Some}(A + B + C(u32))$ memory values

None

$\boxed{0\ldots01}$

Some (A)

$\boxed{\&} \to \boxed{0\ldots0}\boxed{0\ldots01}$

Some (B)

$\boxed{\&} \to \boxed{0\ldots0}\boxed{0\ldots11}$

Some (C (n))

$\boxed{\&} \to \boxed{0\ldots0}\boxed{\&} \to \boxed{0\ldots0}\boxed{n.1}$

To real decision trees!

Combine

- The base decision tree
- Specification of the language

OCaml decision tree

$\text{switch}(v \mathbin{\&} 1)$

$\quad 0 \mapsto \text{switch}((*v).1 \mathbin{\&} 1)$

$\quad\quad 1 \mapsto \text{switch}((*v).1)$

$\quad\quad\quad 01 \mapsto 0, \emptyset$

$\quad\quad\quad 11 \mapsto 1, \emptyset$

$\quad\quad 0 \mapsto 2, \{n \mapsto (*(*v).1).1\}$

$\quad 1 \mapsto 0, \emptyset$

How to implement integer-level switches ?

The CPU doesn't have switches!

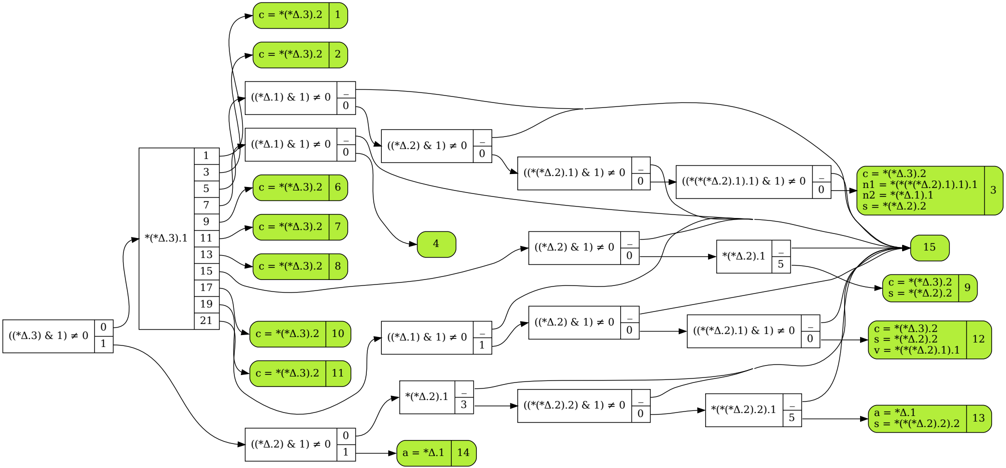
Switch implementation: highly depends on the instruction set:

- If-trees
- Bitmasks
- Jump-tables

Big example

A big example (interpreter for a stack language):

```
matching (a, slist, clist) with
| _ , _, Cons(Ldi, c) -> 1
| _ , _, Cons(Push, c) -> 2
| Int(n2) , Cons (Val (Int (n1)), s), Cons(IOp, c) -> 3
| Int(_) , _, Cons(Test,_) -> 4
| Int(_) , _, Cons(Test,_) -> 5
| _ , _, Cons(Extend,c) -> 6
| _ , _, Cons(Search,c) -> 7
| _ , _, Cons(Pushenv,c) -> 8
| _ , Cons(Env,s), Cons(Popenv,c) -> 9
| _ , _, Cons(Mkclos,c) -> 10
| _ , _, Cons(Mkclosrec,c) -> 11
| Clo , Cons(Val(v),s), Cons(Apply,c) -> 12
| a , Cons(Code,Cons(Env,s)), Nil -> 13
| a , Nil, Nil -> 14
| _ -> 15
```


((*Δ.3) & 1) ≠ 0
*(*Δ.3).1
((*Δ.1) & 1) ≠ 0
((*Δ.2) & 1) ≠ 0
((*(*Δ.2).1) & 1) ≠ 0
((*(*(*Δ.2).1).1) & 1) ≠ 0
*(*Δ.2).1
((*(*Δ.2).2) & 1) ≠ 0
((*Δ.2).2).1
c = *(*Δ.3).2
n1 = *(*(*(*Δ.2).1).1).1
n2 = *(*Δ.1).1
s = *(*Δ.2).2
v = *(*(*Δ.2).1).1
a = *Δ.1
s = *(*(*Δ.2).2).2

Exo time

Let's compile the following pattern matrix with the heuristics "first row that has a pattern":

$$\left(\begin{array}{c|c} \cdot & \\ \hline \langle A, B\rangle & \emptyset, 1 \\ \langle B, \_\rangle & \emptyset, 2 \\ \langle C(x), C(y)\rangle & \emptyset, 3 \\ \langle \_, x\rangle & \emptyset, 4 \end{array}\right)$$

Conclusion

We have seen how to compile pattern matching

- Not so trivial! Lot's of optimization opportunities
- Essential in functional languages
- Also useful elsewhere: LLVM has similar algorithms for cases on strings

Takeaway $\Rightarrow$ "Niche" features deserve their compilation too

A Toy Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(void) {
  char* program;
  int (*fnptr)(void);
  int a;
  program = mmap(NULL, 1000, PROT_EXEC | PROT_READ |
     PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
  program[0] = 0xB8;
  program[1] = 0x34;
  program[2] = 0x12;
  program[3] = 0;
  program[4] = 0;
  program[5] = 0xC3;
  fnptr = (int (*)(void)) program;
  a = fnptr();
  printf("Result = %X\n",a);
}
```

1) What is the program on the left doing?
2) What is **this** API all about?
3) What does this program have to do with a just-in-time compiler?

A Toy Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(void) {
  char* program;
  int (*fnptr)(void);
  int a;
  program = mmap(NULL, 1000, PROT_EXEC | PROT_READ |
     PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
  program[0] = 0xB8;
  program[1] = 0x34;
  program[2] = 0x12;
  program[3] = 0;
  program[4] = 0;
  program[5] = 0xC3;
  fnptr = (int (*)(void)) program;
  a = fnptr();
  printf("Result = %X\n",a);
}
```

A Toy Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(void) {
  char* program;
  int (*fnptr)(void);
  int a;
  program = mmap(NULL, 1000, PROT_EXEC | PROT_READ |
     PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
  program[0] = 0xB8;
  program[1] = 0x34;
  program[2] = 0x12;
  program[3] = 0;
  program[4] = 0;
  program[5] = 0xC3;
  fnptr = (int (*)(void)) program;
  a = fnptr();
  printf("Result = %X\n",a);
}
```

A Toy Example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(void) {
  char* program;
  int (*fnptr)(void);
  int a;
  program = mmap(NULL, 1000, PROT_EXEC | PROT_READ |
     PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
  program[0] = 0xB8;
  program[1] = 0x34;
  program[2] = 0x12;
  program[3] = 0;
  program[4] = 0;
  program[5] = 0xC3;
  fnptr = (int (*)(void)) program;
  a = fnptr();
  printf("Result = %X\n",a);
}
```

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Just-in-Time Compilers

- A JIT compiler translates a program into binary code while this program is being executed.
- We can compile a function as soon as it is necessary.
  - This is Google's V8 approach.
- Or we can first interpret the function, and after we realize that this function is hot, we compile it into binary.
  - This is the approach of Mozilla's IonMonkey.

1) When/where/ why are just-in-time compilers usually used?

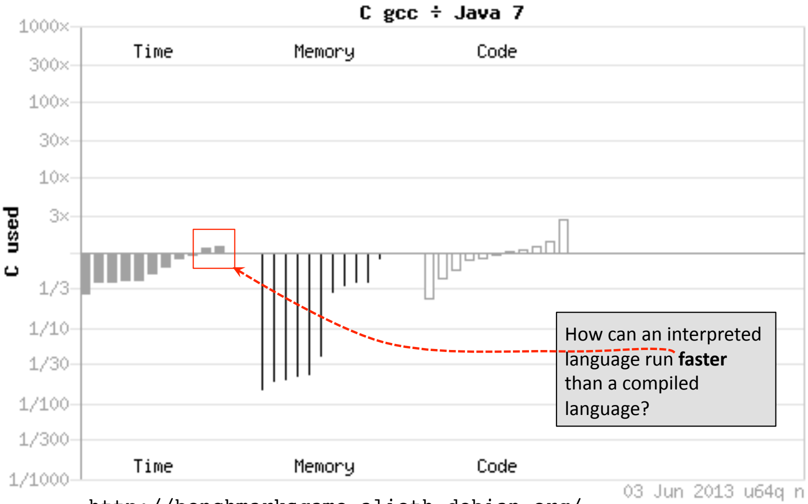
2) Can a JIT compiled program run faster than a statically compiled program?

3) Which famous JIT compilers do we know?

There are many JIT compilers around

- Java Hotspot is one of the most efficient JIT compilers in use today. It was released in 1999, and has been in use since then.
- V8 is the JavaScript JIT compiler used by Google Chrome.
- IonMonkey is the JavaScript JIT compiler used by the Mozilla Firefox.
- LuaJIT (`http://luajit.org/`) is a trace based just-in-time compiler that generates code for the Lua programming language.
- The .Net framework JITs CIL code.
- For Python we have PyPy, which runs on Cpython.

Can JITs compete with static compilers?



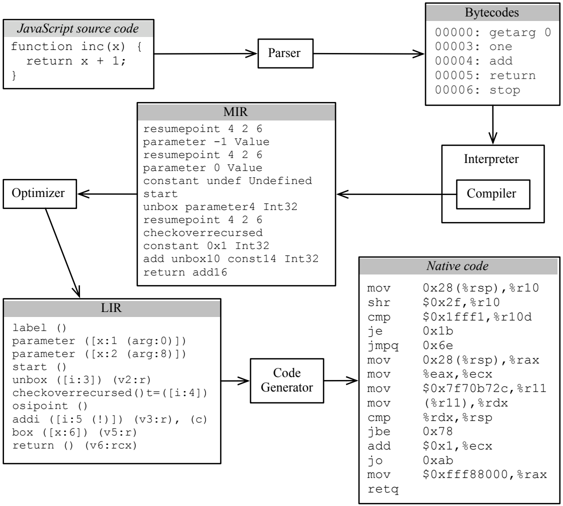
http://benchmarksgame.alioth.debian.org/

Tradeoffs

- There are many tradeoffs involved in the JIT compilation of a program.
- The time to compile is part of the total execution time of the program.
- We may be willing to run simpler and faster optimizations (or no optimization at all) to diminish the compilation overhead.
- And we may try to look at runtime information to produce better codes.
  - Profiling is a big player here.
- The same code may be compiled many times!

Why would we compile the same code many times?

Example: Mozilla's IonMonkey



IonMonkey is one of the JIT compilers used by the Firefox browser to execute JavaScript programs.

This compiler is tightly integrated with SpiderMonkey, the JavaScript interpreter.

SpiderMonkey invokes IonMonkey to JIT compile a function either if it is often called, or if it has a loop that executes for a long time.

Why do we have so many different intermediate representations here?

When to Invoke the JIT Compiler?

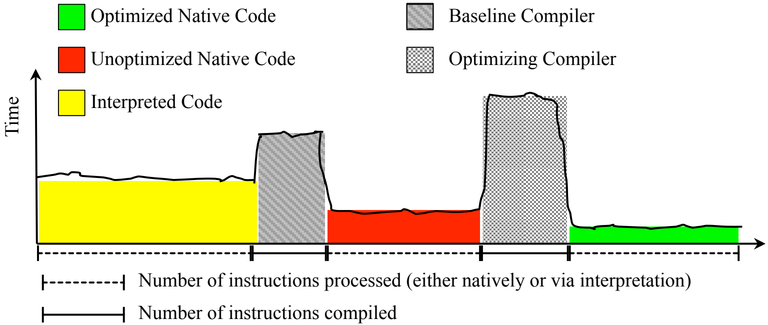
- Compilation has a cost.
  - Functions that execute only once, for a few iterations, should be interpreted.
- Compiled code runs faster.
  - Functions that are called often, or that loop for a long time should be compiled.
- And we may have different optimization levels...

How to decide when to compile a piece of code?

As an example, SpiderMonkey uses three execution modes: the first is interpretation; then we have the baseline compiler, which does not optimize the code. Finally IonMonkey kicks in, and produces highly optimized code.

The Compilation Threshold

Many execution environments associate counters with branches. Once a counter reaches a given threshold, that code is compiled. But defining which threshold to use is very difficult, e.g., how to minimize the area of the curve below? JITs are crowded with magic numbers.



The Million-Dollars Question

- When to invoke the JIT compiler?

1) Can you come up with a strategy to invoke the JIT compiler that optimizes for speed?
2) Do you have to execute the program a bit before calling the JIT?
3) How much information do you need to make a good guess?
4) What is the price you pay for making a wrong prediction?
5) Which programs are easy to predict?
6) Do the easy programs reflect the needs of the users?

3. Just in Time
    - Speculation
    - Tracing

Speculation

- A key trick used by JIT compilers is *speculation*.
- We may assume that a given property is true, and then we produce code that capitalizes on that speculation.
- There are many different kinds of speculation, and they are always a *gamble*:

Speculation

- A key trick used by JIT compilers is *speculation*.
- We may assume that a given property is true, and then we produce code that capitalizes on that speculation.
- There are many different kinds of speculation, and they are always a *gamble*:
  - Let's assume that the type of a variable is an integer,
    - but if we have an integer overflow…
  - Let's assume that the properties of the object are fixed,
    - but if we add or remove something from the object…
  - Let's assume that the target of a call is always the same,
    - but if we point the function reference to another closure…

Inline Caching

- One of the earliest, and most effective, types of specialization was *inline caching*, an optimization developed for the Smalltalk programming language♧.
- Smalltalk is a dynamically typed programming language.
- In a nutshell, objects are represented as *hash-tables*.
- This is a very flexible programming model: we can add or remove properties of objects at will.
- Languages such as Python and Ruby also implement objects in this way.
- Today, inline caching is the key idea behind JITs's high performance when running JavaScript programs.

♧: Efficient implementation of the smalltalk-80 system, POPL (1984)

Using Python Objects

```
def fill(set, b, e, s):
  for i in range(b, e, s):
    set.add(i)

s0 = Set(15)
fill(s0, 10, 20, 3)

s1 = ErrorSet(15)
fill(s1, 10, 14, 3)

class X:
  def __init__(self):
    self.a = 0

fill(X(), 1, 10, 3)
>>>  AttributeError: X instance
>>>  has no attribute 'add'
```

1) What does the function fill do?
2) Why did the third call of fill failed?
3) What are the requirements that fill expects on its parameters?

Duck Typing

```
def fill(set, b, e, s):
  for i in range(b, e, s):
    set.add(i)

class Num:
  def __init__(self, num):
    self.n = num
  def add(self, num):
    self.n += num
  def __str__(self):
    return str(self.n)

n = Num(3)
print n
fill(n, 1, 10, 1)
...
```

Do we get an error **here**?

Duck Typing

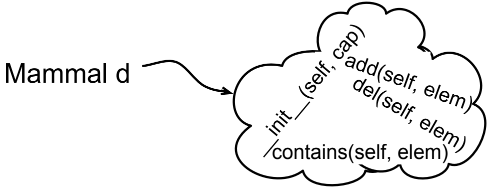
```
class Num:
  def __init__(self, num):
    self.n = num
  def add(self, num):
    self.n += num
  def __str__(self):
    return str(self.n)

n = Num(3)
print n
fill(n, 1, 10, 1)
print n

>>> 3
>>> 48
```

The program works just fine. The only requirement that `fill` expects on its first argument is that it has a method `add` that takes two parameters. Any object that has this method, and can receive an integer on the second argument, will work with `fill`. This is called **duck typing**: *if it quacks like a duck, swims like a duck, eats like a duck, then it is a duck!*

The Price of Flexibility

- Objects, in these dynamically typed languages, are for the most part implemented as hash tables.
  - That is cool: we can add or remove methods without much hard work.
  - And mind how much code we can reuse?
- But method calls are pretty expensive.

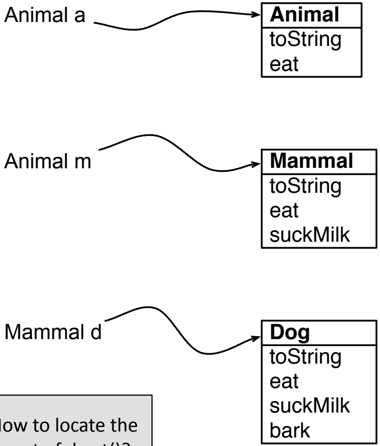
```
def fill(set, b, e, s):
  for i in range(b, e, s):
    set.add(i)
```



How can we make these calls cheaper?

Virtual Tables

```
class Animal {
  public void eat() {
    System.out.println(this + " is eating");
  }
  public String toString () { return "Animal"; }
}

class Mammal extends Animal {
  public void suckMilk() {
    System.out.println(this + " is sucking");
  }
  public String toString () { return "Mammal"; }
  public void eat() {
    System.out.println(this + " is eating like a mammal"); }
}

class Dog extends Mammal {
  public void bark() {
    System.out.println(this + " is barking");
  }
  public String toString () { return "Dog"; }
  public void eat() {
    System.out.println(this + ", is eating like a dog");
  }
}
```

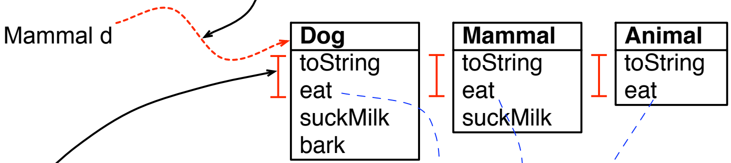


How to locate the target of d.eat()?

Virtual Call

d.eat()

First, we need to know the table d is pointing to. This requires one pointer dereference:



Second, we need to know the offset of the method eat, inside the table. This offset is always the same for any class that inherits from Animal, so we can jump blindly.

```
public void eat() {
  System.out.println ("Eats like a dog");
}

public void eat() {
  System.out.println ("Eats like a mammal");
}

public void eat() {
  System.out.println ("Eats like an animal");
}
```

Can we have virtual tables in duck typed languages?

Monomorphic Inline Cache

```
class Num:
  def __init__(self, n):
    self.n = n
  def add(self, num):
    self.n += num


def fill(set, b, e, s):
  for i in range(b, e, s):
    set.add(i)

n = Num(3)
print n
fill(n, 1, 10, 1)
print n

>>> 3
>>> 48
```

The first time we generate code for a call, we can check the target of that call. *We know this target, because we are generating code at runtime*!

```
fill(set, b, e, s):
 for i in range(b, e, s):
  if isinstance(set, Num):
   set.n += i
  else:
   add = lookup(set, "add")
   add(set, i)
```

Could you optimize this code even further using classic compiler transformations?

Inlining on the Method

- We can also speculate on the method name, instead of doing it on the calling site:

```
fill(set, b, e, s):
  for i in range(b, e, s):
    if isinstance(set, Num):
      set.n += i
    else:
      add = lookup(set, "add")
      add(set, i)
```

```
fill(set, b, e, s):
  for i in range(b, e, s):
    __f_add(set, i)

__f_add(o, e):
  if isinstance(o, Num):
    o.n += e
  else:
    f = lookup(o, "add")
    f(o, e)
```

1) Is there any advantage to this approach, when compared to inlining at the call site?
2) Is there any disadvantage?
3) Which one is likely to change more often?

Polymorphic Calls

- If the target of a call changes during the execution of the program, then we have a polymorphic call.
- A monomorphic inline cache would have to be invalidated, and we would fall back into the expensive quest.

```
>>> l = [Num(1), Set(1), Num(2), Set(2), Num(3), Set(3)]
>>> for o in l:
...   o.add(2)
...
```

Is there anything we could do to optimize this case?

Polymorphic Calls

- If the target of a call changes during the execution of the program, then we have a polymorphic call.
- A monomorphic inline cache would have to be invalidated, and we would fall back into the expensive quest.

```
>>> l = [Num(1), Set(1), Num(2), Set(2),
Num(3), Set(3)]
>>> for o in l:
...    o.add(2)
...
```

Would it not be better just to have the code below?

```
for i in range(b, e, s):
   lookup(set, "add")
   ...
```

```
fill(set, b, e, s):
   for i in range(b, e, s):
      __f_add(set, i)

__f_add(o, e):
   if isinstance(o, Num):
      o.n = e
   elif isinstance(o, Set):
      (index, bit) = getIndex(e)
      o.vector[index] |= bit
   else:
      f = lookup(o, "add")
      f(o, e)
```

The Speculative Nature of Inline Caching

- Python – as well as JavaScript, Ruby, Lua and other very dynamic languages – allows the user to add or remove methods from an object.
- If such changes in the layout of an object happen, then the representation of that object must be recompiled. In this case, we need to update the inline cache.

```
from Set import INT_BITS, getIndex, Set

def errorAdd(self, element):
  if (element > self.capacity):
    raise IndexError(str(element) +
      " is out of range.")
  else:
    (index, bit) = getIndex(element)
    self.vector[index] |= bit
    print element, "added successfully!"

Set.add = errorAdd
s = Set(60)
s.errorAdd(59)
s.remove(59)
```

The Benefits of the Inline Cache

These numbers have been obtained by Ahn *et al*. for JavaScript, in the Chrome V8 compiler♤:

- Monomorphic inline cache hit:
  - 10 instructions
- Polymorphic Inline cache hit:
  - 35 instructions if there are 10 types
  - 60 instructions if there are 20 types
- Inline cache miss: 1,000 – 4,000 instructions.

Which factors could justify these numbers?

♤: Improving JavaScript Performance by Deconstructing the Type System, PLDI (2014)

3 Just in Time
- Speculation
- Tracing

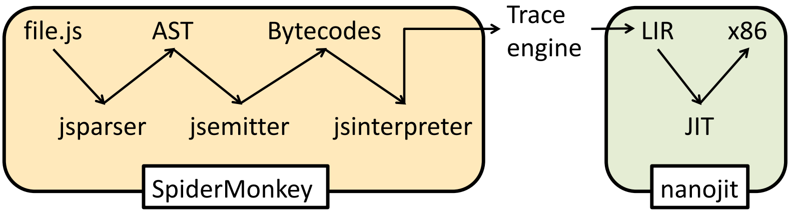
What is a JIT trace compiler?

- A trace-based JIT compiler translates only the most executed paths in the program's control flow to machine code.
- A trace is a linear sequence of code, that represents a hot path in the program.
- Two or more traces can be combined into a tree.
- Execution alternates between traces and interpreter.

What are the advantages and disadvantages of trace compilation over traditional method compilation?

The anatomy of a trace compiler

- TraceMonkey is the trace based JIT compiler used in the Mozilla Firefox Browser.



From source to bytecodes

```
function foo(n){
  var sum = 0;
  for(i = 0; i < n; i++){
   sum+=i;
  }
  return sum;
}
```

jsparser → AST → jsemitter

Can you see the correspondence between source code and bytecodes?

```
00: getname n
02: setlocal 0
06: zero
07: setlocal 1
11: zero
12: setlocal 2
16: goto 35 (19)
19: trace
20: getlocal 1
23: getlocal 2
26: add
27: setlocal 1
31: localinc 2
35: getlocal 2
38: getlocal 0
41: lt
42: ifne 19 (-23)
45: getlocal 1
48: return
49: stop
```

The trace engine kicks in

- TraceMonkey interprets the bytecodes.
- Once a **loop** is found, it may decide to ask Nanojit to transform it into machine code (e.g, x86, ARM).
  - Nanojit reads LIR and produces x86
- Hence, TraceMonkey must convert **this** trace of bytecodes into LIR

```
00: getname n
02: setlocal 0
06: zero
07: setlocal 1
11: zero
12: setlocal 2
16: goto 35 (19)
19: trace
20: getlocal 1
23: getlocal 2
26: add
27: setlocal 1
31: localinc 2
35: getlocal 2
38: getlocal 0
41: lt
42: ifne 19 (-23)
45: getlocal 1
48: return
49: stop
```

Bytecodes

From bytecodes to LIR

```
00: getname n
02: setlocal 0
06: zero
07: setlocal 1
11: zero
12: setlocal 2
16: goto 35 (19)
19: trace
20: getlocal 1
23: getlocal 2
26: add
27: setlocal 1
31: localinc 2
35: getlocal 2
38: getlocal 0
41: lt
42: ifne 19 (-23)
45: getlocal 1
48: return
49: stop
```

```
L: load "i" %r1
   load "sum" %r2
   add %r1 %r2 %r1
   %p0 = ovf()
   bra %p0 Exit1
   store %r1 "sum"
   inc %r2
   store %r2 "i"
   %p0 = ovf()
   bra %p0 Exit2
   load "i" %r0
   load "n" %r1
   lt %p0 %r0 %r1
   bne %p0 L
```

Nanojit LIR

From bytecodes to LIR

Bytecodes

```
00: getname n
02: setlocal 0
06: zero
07: setlocal 1
11: zero
12: setlocal 2
16: goto 35 (19)
19: trace
20: getlocal 1
23: getlocal 2
26: add
27: setlocal 1
31: localinc 2
35: getlocal 2
38: getlocal 0
41: lt
42: ifne 19 (-23)
45: getlocal 1
48: return
49: stop
```

Nanojit LIR

```
L:  load "i" %r1
    load "sum" %r2
    add %r1 %r2 %r1
    %p0 = ovf()
    bra %p0 Exit1
    store %r1 "sum"
    inc %r2
    store %r2 "i"
    %p0 = ovf()
    bra %p0 Exit2
    load "i" %r0
    load "n" %r1
    lt %p0 %r0 %r1
    bne %p0 L
```

Overflow tests are required by operations such as add, sub, inc, dec, mul.

Why do we have overflow tests?

- Many scripting languages represent numbers as floating-point values.
  - Arithmetic operations are not very efficient.
- The compiler sometimes is able to infer that these numbers can be used as integers.
  - But floating-point numbers are larger than integers.
  - This is another example of speculative optimization.
  - Thus, every arithmetic operation that might cause an overflow must be preceded by a test. If the test fails, then the runtime engine must change the number's type back to floating-point.

From LIR to assembly

```
L:  load "i" %r1
    load "sum" %r2
    add %r1 %r2 %r1
    %p0 = ovf()
    bra %p0 Exit1
    store %r1 "sum"
    inc %r2
    store %r2 "i"
    %p0 = ovf()
    bra %p0 Exit2
    load "i" %r0
    load "n" %r1
    lt %p0 %r0 %r1
    bne %p0 L
```

Nanojit LIR

The overflow tests are also translated into machine code.

Can you come up with an optimization to eliminate some of the overflow checks?

x86 Assembly

```
L:movl   -32(%ebp), %eax
  movl   %eax, -20(%ebp)
  movl   -28(%ebp), %eax
  movl   %eax, -16(%ebp)
  movl   -20(%ebp), %edx
  leal   -16(%ebp), %eax
  addl   %edx, (%eax)
  call   _ovf
  testl  %eax, %eax
  jne    Exit1
  movl   -16(%ebp), %eax
  movl   %eax, -28(%ebp)
  leal   -20(%ebp), %eax
  incl   (%eax)
  call   _ovf
  testl  %eax, %eax
  jne    Exit2
  movl   -24(%ebp), %eax
  movl   %eax, -12(%ebp)
  movl   -20(%ebp), %eax
  cmpl   -12(%ebp), %eax
  jl     L
```

How to eliminate the redundant tests

- We use range analysis:
  - Find the range of integer values that a variable might hold during the execution of the trace.

```
function foo(n){
  var sum = 0;
  for(i = 0; i < n; i++){
   sum+=i;
  }
  return sum;
}
```

Example: if we know that n is 10, and i is always less than n, then we will never have an overflow if we add 1 to i.

Cheating at runtime

- A **static analysis** must be very conservative: if we do not know for sure the value of n, then we must assume that it may be anywhere in [-∞, +∞].
- **However, we are not a static analysis!**

```
function foo(n){
  var sum = 0;
  for(i = 0; i < n; i++){
   sum+=i;
  }
  return sum;
}
```

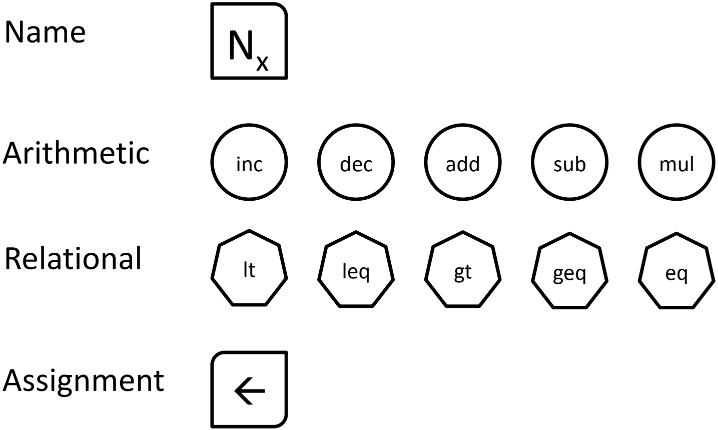
- We are compiling at runtime!
- To know the value of n, just ask the interpreter.

How the algorithm works

- Create a constraint graph.
  - While the trace is translated to LIR.
- Propagate range intervals.
  - Before sending the LIR to Nanojit.
  - Using infinite precision arithmetic.
- Eliminate tests whenever it is safe to do so.
  - We tell Nanojit that code for some overflow tests should not be produced.

The constraint graph

- We have four categories of vertices:



Building the constraint graph

19: trace
20: getlocal *sum*
23: getlocal *i*
26: add
27: setlocal *sum*
31: localinc *i*
35: getlocal *n*
38: getlocal *i*
41: lt
42: ifne 19 (-23)

- We start building the constraint graph once TraceMonkey starts recording a trace.
- TraceMonkey starts at the branch instruction, which is the first instruction visited in the loop.
  - Although it is at the end of the trace.

In terms of code generation, can you recognize the pattern of bytecodes created for the test if n < i goto L?

[10, 10]

n_0

19: trace
20: getlocal *sum*
23: getlocal *i*
26: add
27: setlocal *sum*
31: localinc *i*
35: getlocal *n*
38: getlocal *i*
41: lt
42: ifne 19 (-23)

We check the interpreter's stack to find out that n is 10.

[10, 10] n_0

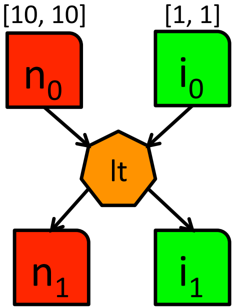
[1, 1] i_0

i started holding 0, but at this point, its value is already 1.

19: trace
20: getlocal *sum*
23: getlocal *i*
26: add
27: setlocal *sum*
31: localinc *i*
35: getlocal *n*
38: getlocal *i*
41: lt
42: ifne 19 (-23)

Why we do not initialize i with 0 in our constraint graph?

19: trace
20: getlocal *sum*
23: getlocal *i*
26: add
27: setlocal *sum*
31: localinc *i*
35: getlocal *n*
38: getlocal *i*
41: lt
42: ifne 19 (-23)

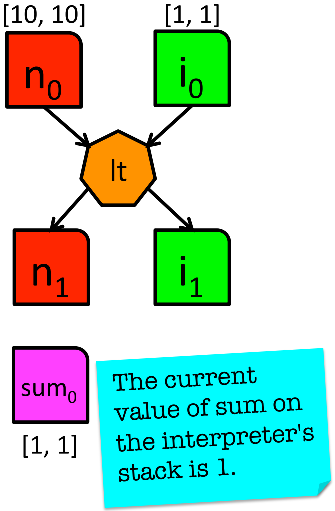


Comparisons are great! We can learn new information about variables

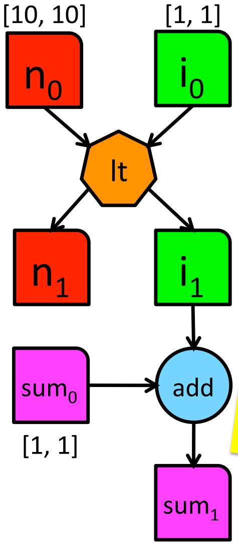
Now we know that i is less than 10, so we rename it to i_1 we also rename n

We are renaming after conditionals. Which program representation are we creating *dynamically*?

19: trace
20: getlocal *sum*
23: getlocal *i*
26: add
27: setlocal *sum*
31: localinc *i*
35: getlocal *n*
38: getlocal *i*
41: lt
42: ifne 19 (-23)



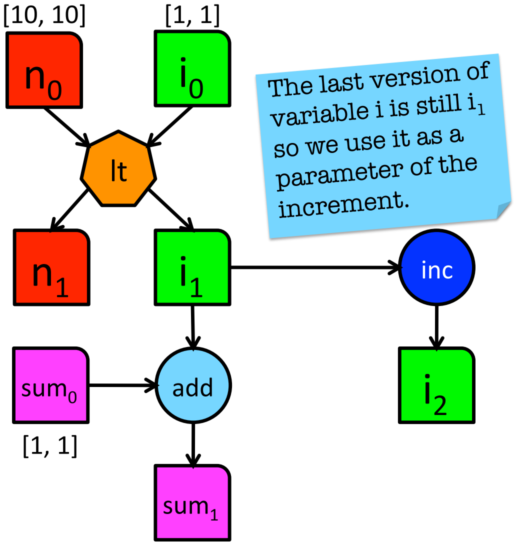
19: **trace**
20: **getlocal *sum***
23: **getlocal *i***
26: **add**
27: **setlocal *sum***
31: **localinc *i***
35: **getlocal *n***
38: **getlocal *i***
41: **lt**
42: **ifne 19 (-23)**

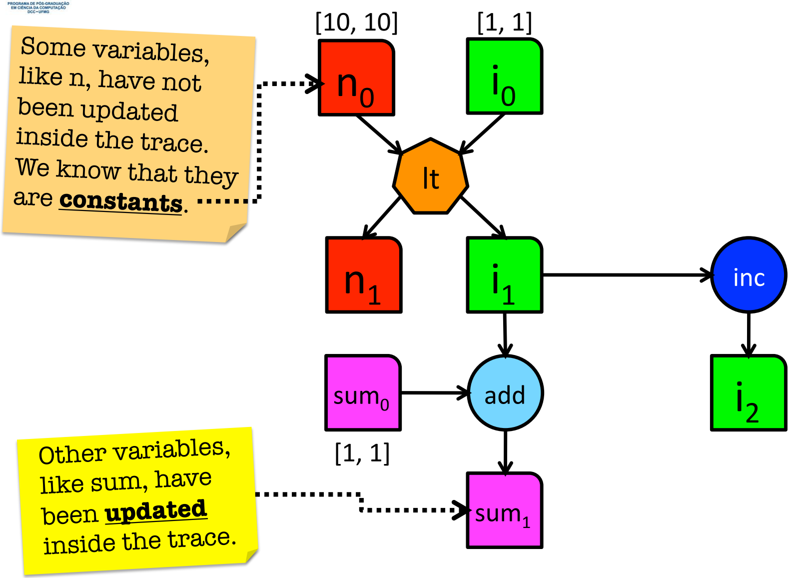


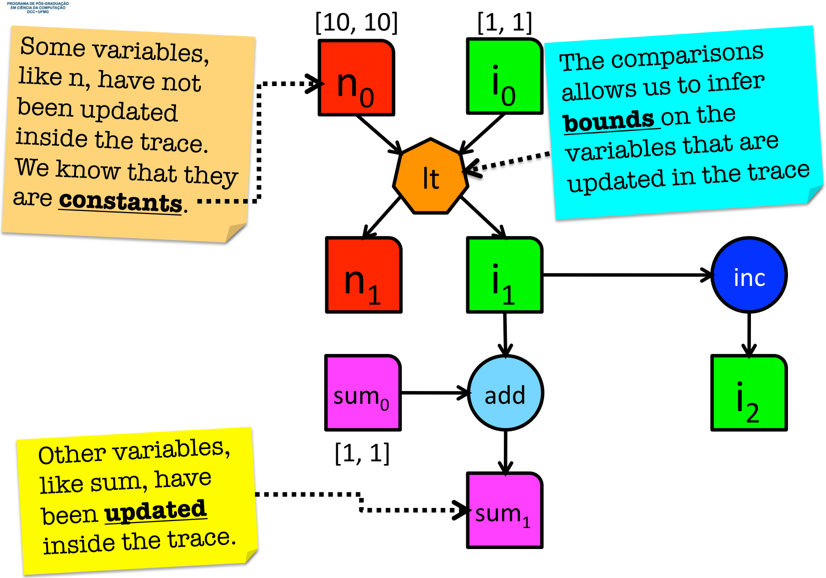
Why do we pair up sum_0 with i_1, and not with i_0?

For the sum, we get the last version of variable i, which is i_1.

19: trace
20: getlocal *sum*
23: getlocal *i*
26: add
27: setlocal *sum*
31: localinc *i*
35: getlocal *n*
38: getlocal *i*
41: lt
42: ifne 19 (-23)

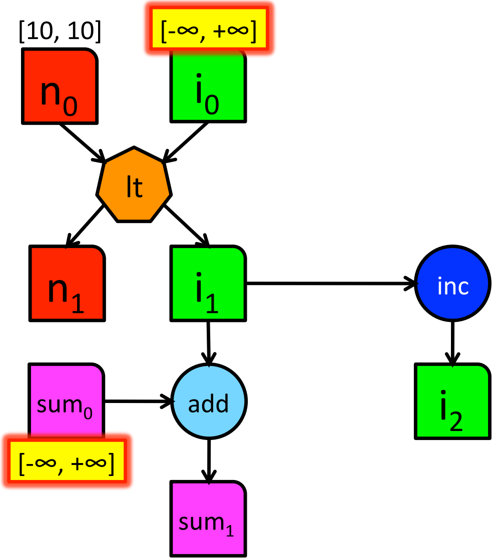


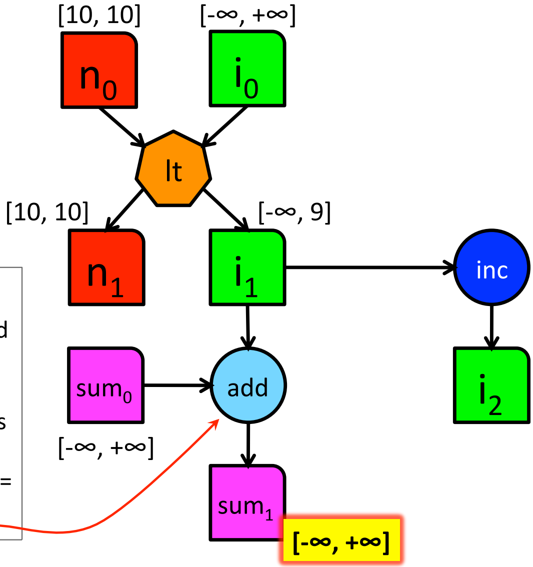

Some variables, like n, have not been updated inside the trace. We know that they are **constants**.
[10, 10]
[1, 1]
n0
i0
lt
n1
i1
inc
i2
sum0
add
[1, 1]
sum1
Other variables, like sum, have been **updated** inside the trace.


Some variables, like n, have not been updated inside the trace. We know that they are **constants**.
[10, 10]
[1, 1]
n_0
i_0
The comparisons allows us to infer **bounds** on the variables that are updated in the trace
lt
n_1
i_1
inc
sum_0
add
i_2
[1, 1]
Other variables, like sum, have been **updated** inside the trace.
sum_1

The next phase of our algorithm is the propagation of range intervals.

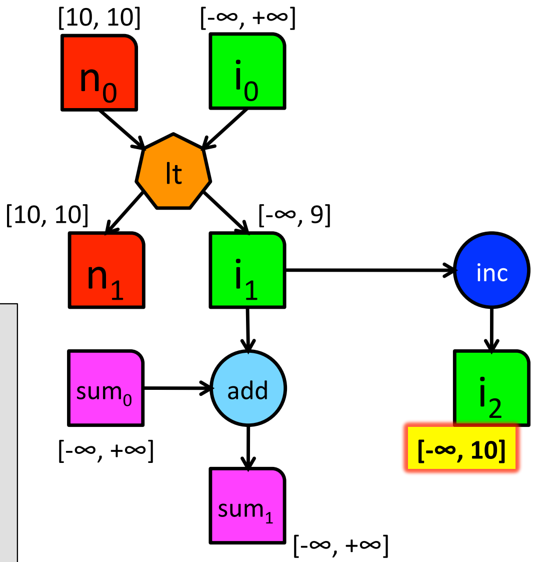
We start by assigning **conservative** i.e, [-∞, +∞], bounds to the ranges of variables updated inside the trace.





We are doing abstract interpretation. We need an abstract semantics for every operation in our constraint graph. As an example, we know that [-∞, +∞] + [-∞, 9] = [-∞, +∞]

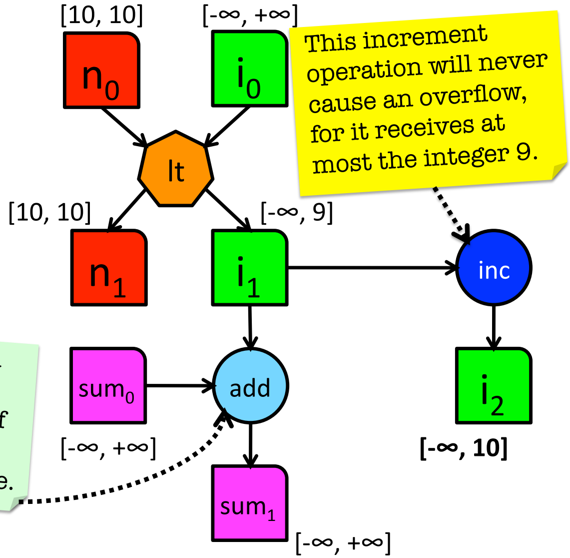
After range propagation we can check which overflow tests are really necessary.



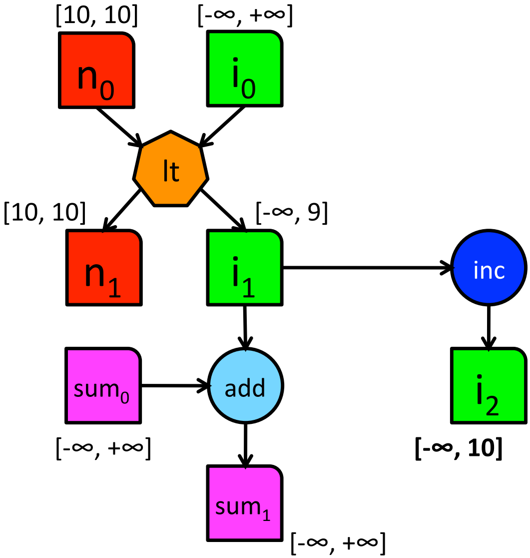
1) How many overflow checks do we have in this constraint graph?

2) Is there any overflow check that is redundant?

After range propagation we can check which overflow tests are really necessary.



```
L: load "i" %r1
   load "sum" %r2
   add %r1 %r2 %r1
   %p0 = ovf()
   bra %p0 Exit1
   store %r1 "sum"
   inc %r2
   store %r2 "i"
   %p0 = ovf()
   bra %p0 Exit2
   load "i" %r0
   load "n" %r1
   lt %p0 %r0 %r1
   bne %p0 L
```



JIT – Conclusion

Just-In-Time compilation combines dynamic runtime information with static compilation

Practical approach: Use whatever available to make it fast.

The most used compilers in the world are Web Browsers!

Further in Compilation

Many other “compilations” that what we have seen: dynamic languages; objects, functional, and other paradigms; Data manipulation, and other stranger computations mode (see DM from previous years!).

Recent “fun” example: implementation of the French Tax System.

Domain Specific Languages: the new frontier

Different domains have very different computations: meteo simulations, genome analysis, cryptography, 3D rendering, Machine Learning, . . .

$\Rightarrow$ The current frontier: How to provide nice languages and efficient compilations for these varied use-cases?