A Functional Perspective
on
SSA Optimization Algorithms

Patryk Zadarnowski

jointly with

Manuel M. T. Chakravarty

Gabriele Keller

COCV'03

Warsaw

12 April 2003

University of New South Wales

Sydney

The Plan

① Motivation — an overview of the *status quo*

② Translating the *static single assignment* (SSA) form into the *administrative normal form* (ANF) of lambda calculus

③ Applying the technique to the *sparse conditional constant propagation* (SCC) algorithm

MOTIVATION

- ➜ We need a verified optimizing compiler
- ➜ Optimization algorithms are:
  - ✗ complex
  - ✗ inadequately specified
- ➜ Fix the data structures before fixing the algorithms!

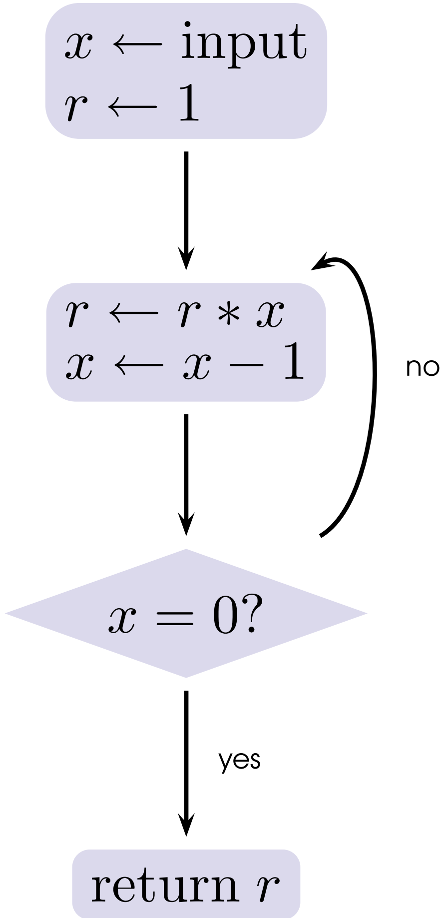
Contributions:

① Formalize the correspondance between the *Static Single Assignment* form and a direct-style lambda calculus

② Redefine Wegman-Zadeck's *Sparse Conditional Constant Propagation* using our intermediate representation

③ Formally establish soundness of the new algorithm

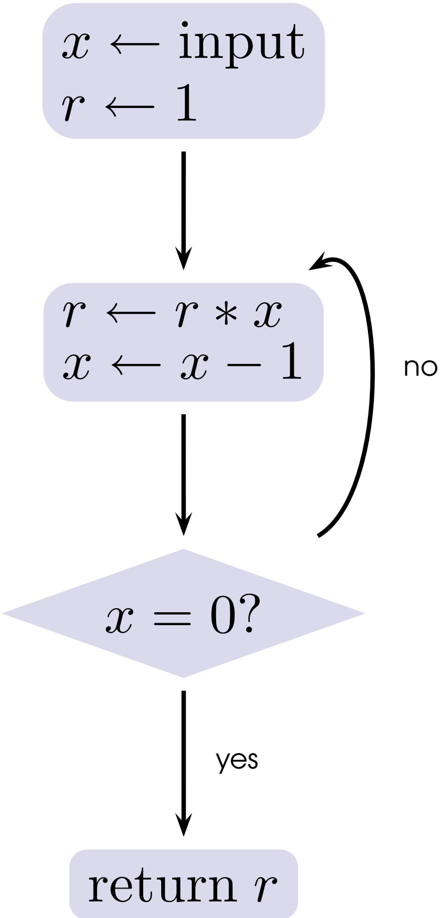
Control Flow Graphs



Summary:

- Low-level representation
- Informal semantics
- Emphasis on *control flow*

Control Flow Graphs



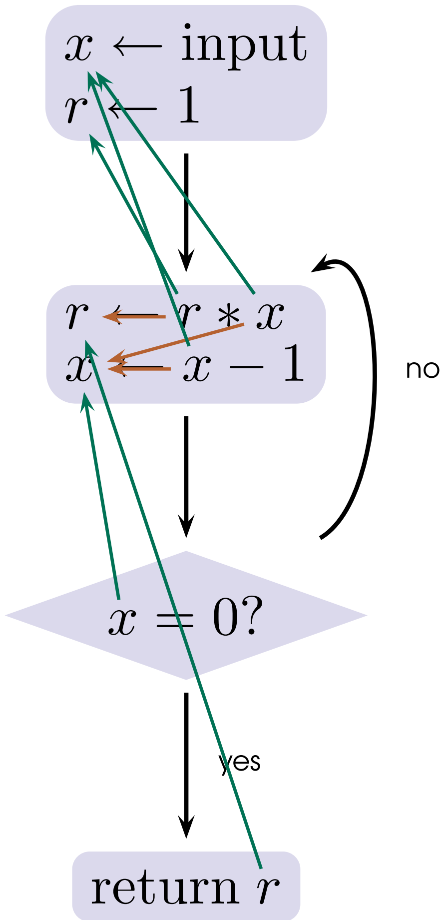
Summary:

- Low-level representation
- Informal semantics
- Emphasis on *control flow*

However, many (most?) optimizations require *data flow* information:

- Expensive to compute
- Reusable

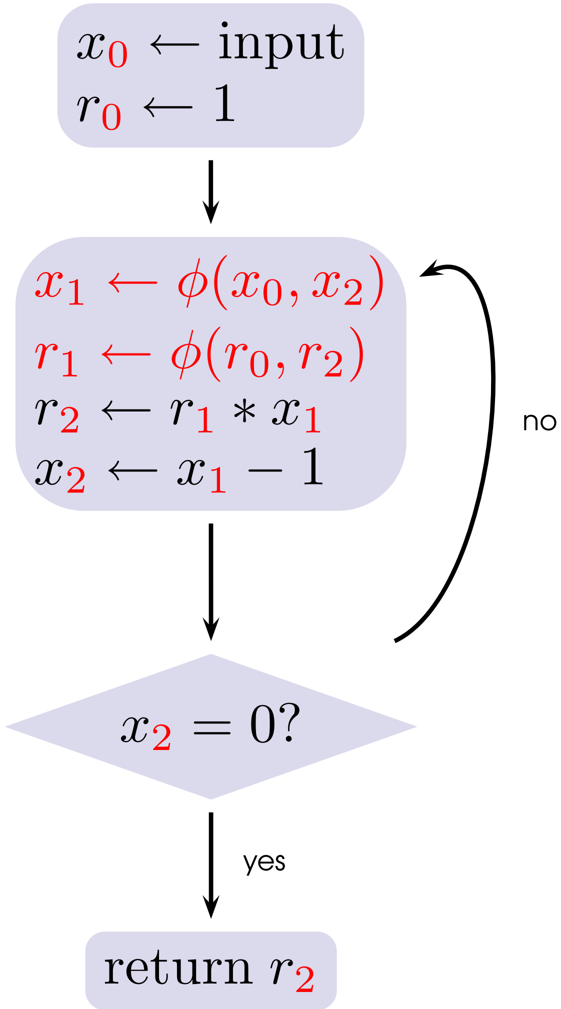
Control Flow Graphs & dataflow information



Summary:

- Def-Use chains: defs refer to uses and uses refer to defs
- Analyses need a *dense* mapping of variables to abstract values
- ✗ Expensive to represent and update!

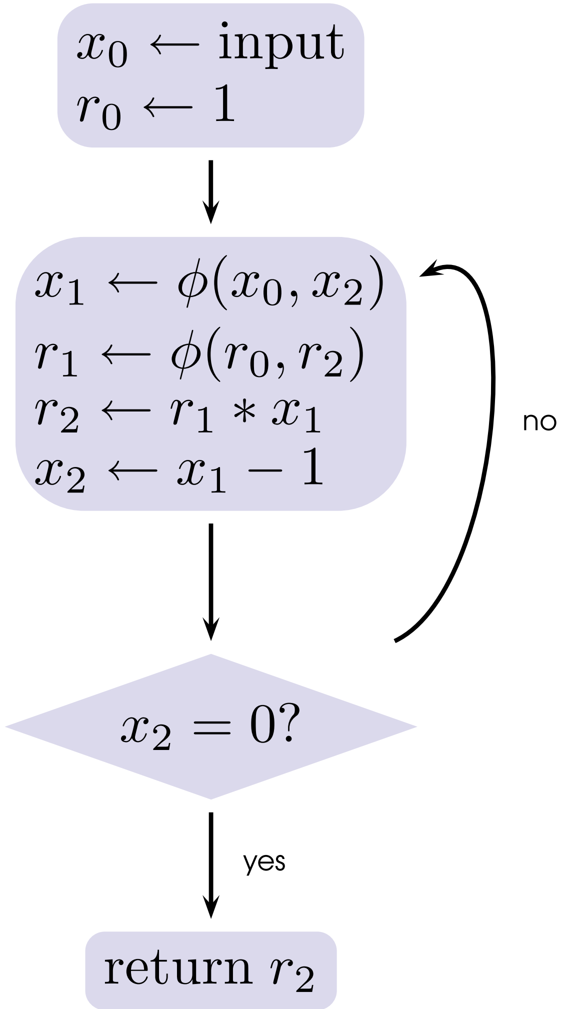
Static Single Assignment Form



Overview:

- Each variable has exactly one defining occurence
- Values from different control flows merged via ϕ functions
- ϕ functions placed by computing the *dominator tree* of the procedure

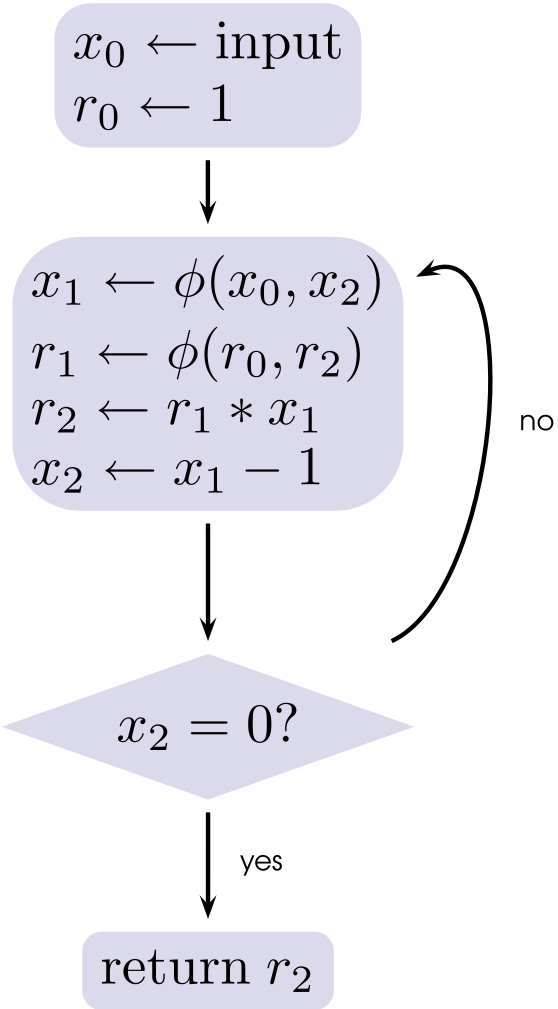
Static Single Assignment Form



Evaluation I:

- ☑ Sparse representation
- ☑ Split variables improve the accuracy of analyses
- ☑ Significantly reduces complexity of optimizations, such as:
  - constant propagation
  - partial redundancy elimination
  - value numbering
  - ...

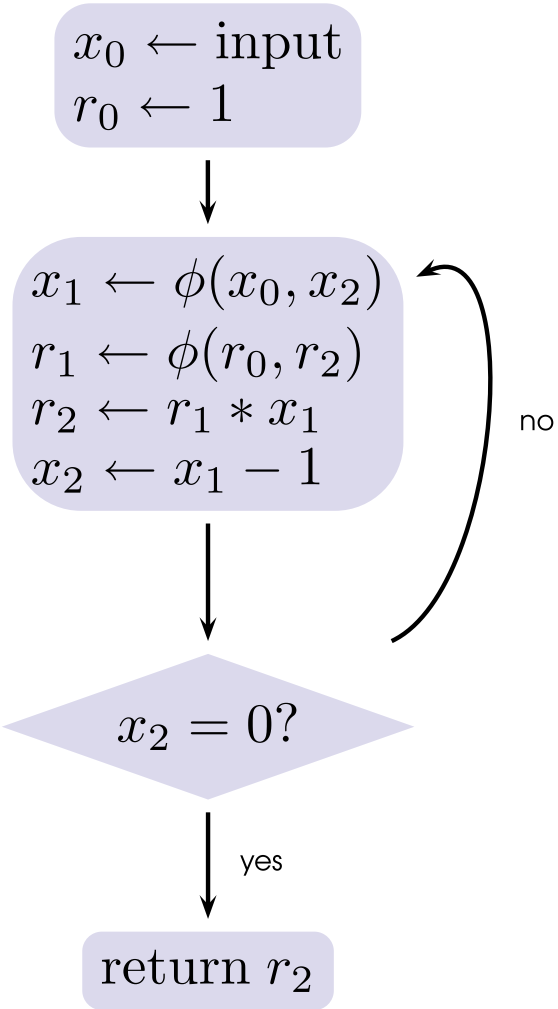
Static Single Assignment Form



Evaluation II:

- ✗ Still no scoping
- ✗ Still only informal semantics
- ✗ Difficult to add a type system:

Static Single Assignment Form



Evaluation II:

- ✗ Still no scoping
- ✗ Still only informal semantics
- ✗ Difficult to add a type system:
  - ϕ parameters out of context
  - Analysis requires a large environment
- ☑ However, there are improved versions, such as Gated SSA

In every FORTRAN program
hides a functional program trying to get out

ADMINISTRATIVE NORMAL FORM

Restricted direct-style lambda calculus:

- ➜ no nested `let`'s
- ➜ no nested function applications
- ➜ no anonymous lambda expressions

Administrative Normal Form

Syntax:

$$e \;::=$$

Administrative Normal Form

Syntax:

$e \ ::= \ \texttt{let}\ x = v\ \texttt{in}\ e$

- copy

Administrative Normal Form

$$
\begin{aligned}
e \; ::= \; & \texttt{let}\ x = v\ \texttt{in}\ e \quad | \\
& \texttt{let}\ x = v(\bar{v})\ \texttt{in}\ e
\end{aligned}
$$

Syntax:

- copy
- calls

Administrative Normal Form

$$
\begin{aligned}
e \; ::= \; & \texttt{let}\ x = v\ \texttt{in}\ e \quad | \\
& \texttt{let}\ x = v(\bar{v})\ \texttt{in}\ e \quad | \\
& v
\end{aligned}
$$

Syntax:

- copy
- calls
- returns

Administrative Normal Form

$$
\begin{aligned}
e \;::=\;& \texttt{let}\ x = v\ \texttt{in}\ e \quad | \\
& \texttt{let}\ x = v(\bar{v})\ \texttt{in}\ e \quad | \\
& v \quad | \\
& v(\bar{v})
\end{aligned}
$$

Syntax:

- copy
- calls
- returns
- jumps

Administrative Normal Form

$$
\begin{array}{lll}
e & ::= & \texttt{let}\ x = v\ \texttt{in}\ e \quad | \\
 & & \texttt{let}\ x = v(\bar{v})\ \texttt{in}\ e \quad | \\
 & & v \quad | \\
 & & v(\bar{v}) \quad | \\
 & & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

Syntax:

- copy
- calls
- returns
- jumps
- branches

Administrative Normal Form

$$
\begin{array}{lcl}
e & ::= & \texttt{let}\ x = v\ \texttt{in}\ e \quad | \\
& & \texttt{let}\ x = v(\bar{v})\ \texttt{in}\ e \quad | \\
& & v \quad | \\
& & v(\bar{v}) \quad | \\
& & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad | \\
& & \texttt{letrec}\ \bar{f}\ \texttt{in}\ e \\
f & ::= & x(\bar{x}) = e
\end{array}
$$

Syntax:

- copy
- calls
- returns
- jumps
- branches
- code labels

Administrative Normal Form

$$
\begin{array}{rcl}
e & ::= & \texttt{let}\ x = v\ \texttt{in}\ e \quad | \\
  &   & \texttt{let}\ x = v(\bar{v})\ \texttt{in}\ e \quad | \\
  &   & v \quad | \\
  &   & v(\bar{v}) \quad | \\
  &   & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad | \\
  &   & \texttt{letrec}\ \bar{f}\ \texttt{in}\ e \\
f & ::= & x(\bar{x}) = e
\end{array}
$$

Evaluation:

- ☑ Natural scoping
- ☑ Clean operational semantics
- ☑ Easily type-checked
- ☑ Yet still close to assembly language!

Administrative Normal Form

So where do I get some?

Administrative Normal Form

So where do I get some?

Translate an SSA program!

- Semi-formal correspondence between programs in SSA form and CPS (Kelsey, 1995)
- Semi-formal translation from SSA form into lambda calculus (Appel, 1998)
- We present a formal translation from SSA form to ANF (based on dominator trees)

Structured SSA Form

```
proc fac(x) {
    goto L₁
L₁: {
        r₀ ← φ(start:1, L₁:r₁)
        x₀ ← φ(start:x, L₁:x₁)
        if x₀ then goto L₁ else ret r₀
    L₂: r₁ ← mul(r₀, x₀)
        x₁ ← sub(x₀, 1)
        goto L₁
    }
}
...
```

Translating Structured SSA to ANF

```
proc fac(x) {

}
...
```

```
letrec fac(x) =

in
   ...
```

Translating Structured SSA to ANF

```
proc fac(x) {

L1:{




  L2:



   }
}
...
```

```
letrec fac(x) =
   letrec fac'(       ) =
     letrec fac''() =




     in


   in


in
   ...
```

Translating Structured SSA to ANF

```
proc fac(x) {

L₁:{

     if x₀              else ret r₀

  L₂:r₁ ← mul(r₀, x₀)
     x₁ ← sub(x₀, 1)

   }
}
...
```

```
letrec fac(x) =
   letrec fac'(      ) =
     letrec fac"() =
       let r₁ = mul(r₀, x₀)
       let x₁ = sub(x₀, 1)
       in
     in
       if x₀          else r₀
   in

in
   ...
```

Translating Structured SSA to ANF

```
proc fac(x) {
   goto L₁
L₁:{



     if x₀ goto L₁ else ret r₀

  L₂:r₁ ← mul(r₀, x₀)
     x₁ ← sub(x₀, 1)
     goto L₁
   }
}
...
```

```
letrec fac(x) =
   letrec fac'(       ) =
     letrec fac''() =
       let r₁ = mul(r₀, x₀)
       let x₁ = sub(x₀, 1)
       in fac'(       )
     in
       if x₀ fac''() else r₀
   in
     fac'(     )
in
   ...
```

Translating Structured SSA to ANF

```
proc fac(x) {
   goto L₁
L₁:{
     r₀ ← φ(               )
     x₀ ← φ(               )
     if x₀ goto L₁ else ret r₀
  L₂:r₁ ← mul(r₀, x₀)
     x₁ ← sub(x₀, 1)
     goto L₁
   }
}
...
```

```
letrec fac(x) =
   letrec fac′(x₀, r₀) =
     letrec fac″() =
       let r₁ = mul(r₀, x₀)
       let x₁ = sub(x₀, 1)
       in fac′(      )
     in
       if x₀ fac″() else r₀
   in
     fac′(    )
in
   ...
```

Translating Structured SSA to ANF

```
proc fac(x) {
  goto L₁
L₁:{
    r₀ ← φ(start:1, L₁:r₁)
    x₀ ← φ(start:x, L₁:x₁)
    if x₀ goto L₁ else ret r₀
  L₂:r₁ ← mul(r₀, x₀)
    x₁ ← sub(x₀, 1)
    goto L₁
  }
}
...
```

```
letrec fac(x) =
  letrec fac'(x₀, r₀) =
    letrec fac''() =
      let r₁ = mul(r₀, x₀)
      let x₁ = sub(x₀, 1)
      in fac'(r₁, x₁)
    in
      if x₀ fac''() else r₀
  in
    fac'(x, 1)
in
  ...
```

Formal definition presented in the paper.

WHAT HAVE WE GAINED?

- ➜ A way of tricking hackers into writing functional programs
- ➜ A way of thinking about functional programs as SSA programs
  - applying SSA algorithms to functional programs
- ➜ A way of thinking about SSA programs as functional programs
  - applying formal techniques to SSA programs

So, let's have a look at a concrete algorithm. . .

Sparse Conditional Constant Propagation

```
proc seven(x) {
   goto L₁
L₁:x₀ ← φ(start:1, L₁:x₁)
   x₁ ← sub(x₀, 1)
   if x₁ then
     goto L₁
   else
     ret 7
}
```

$\longrightarrow$

```
proc seven(x) {
   ret 7
}
```

The original SSA algorithm (Wegman & Zadeck, 1991) rather informally presented. . .

```
i ← 1
...
if i = 1
  then j ← 1
  else j ← 2
```

Fig. 9. A conditional constant definition.

Many optimizing compilers repeatedly execute constant propagation and unreachable code elimination since each provides information that improves the other. CC solves this problem in an elegant way by combining the two optimizations. Additionally, the algorithm gets better results than are possible by repeated applications of the separate algorithms, as described in Section 5.1.

3.4 Sparse Conditional Constant

We wish to derive a version of CC that also improves running time, just as SSC was derived from SC to improve running time. In order to do this, we must utilize some of the special properties of the SSA graph. We call this algorithm *Sparse Conditional Constant* or SCC.

When the SSA graph was constructed, ϕ-functions were inserted at some join nodes. The meaning of a ϕ-function is that if control reaches the node in the program flow graph along its ith in-edge, the result of the ϕ-function is the value of its ith operand.

In the SSC algorithm, when the meet rule was applied to a ϕ-function, the meet operator was applied to all of the operands of the ϕ-function. In the SCC algorithm, the meet operator is applied only to those operands of the ϕ-function that correspond to the program flow graph edges marked executable. Those that are not executable effectively have the value of $\top$.

This algorithm uses two worklists: *FlowWorkList* is a worklist of program flow graph edges and *SSAWorkList* is a worklist of SSA edges.

SCC works as follows:

(1) Initialize the FlowWorkList to contain the edges exiting the start node of the program. The SSAWorkList is initially empty.

Each program flow graph edge has an associated flag, the *ExecutableFlag*, that controls the evaluation of ϕ-functions in the destination node of that edge. This flag is initially `false` for all edges.

Each LatticeCell is initially $\top$.

(2) Halt execution when both worklists become empty. Execution may proceed by processing items from either worklist.

(3) If the item is a program flow graph edge from the FlowWorkList, then examine the ExecutableFlag of that edge. If the ExecutableFlag is `true` do nothing; otherwise:

(a) Mark the ExecutableFlag of the edge as `true`.

(b) Perform Visit-ϕ for all of the ϕ-functions at the destination node.

(c) If only one of the ExecutableFlags associated with the incoming program flow graph edges is `true` (i.e., if this is the first time this

node has been evaluated), then perform VisitExpression for the expression in this node.

(d) If the node only contains one outgoing flow graph edge, add that edge to the FlowWorkList.

(4) If the item is an SSA edge from the SSAWorkList and the destination of that edge is a ϕ-function, perform Visit-ϕ.

(5) If the item is an SSA edge from the SSAWorkList and the destination of that edge is an expression, then examine ExecutableFlags for the program flow edges reaching that node. If any of them are `true`, perform VisitExpression. Otherwise do nothing.

The value of the LatticeCell associated with the output of a ϕ-function is defined to be the meet of all arguments whose corresponding in-edge has been marked executable. It is computed by Visit-ϕ. Visit-ϕ is called whenever the value of the LatticeCell associated with one of its operands is lowered or when the ExecutableFlag associated with one of the in-edges becomes `true`.

Visit-ϕ is defined as follows: The LatticeCells for each operand of the ϕ-function are defined on the basis of the ExecutableFlag for the corresponding program flow edge.

`executable` The LatticeCell has the same value as the LatticeCell at the definition end of the SSA edge.

`not-executable` The LatticeCell has the value $\top$.

VisitExpression is defined as follows: Evaluate the expression obtaining the values of the operands from the LatticeCells where they are defined and using the expression rules defined in Section 2.2. If this changes the value of the LatticeCell of the output of the expression, do the following:

(1) If the expression is part of an assignment node, add to the SSAWorkList all SSA edges starting at the definition for that node.

(2) If the expression controls a conditional branch, some outgoing flow graph edges must be added to the FlowWorkList. If the LatticeCell has value $\bot$, all exit edges must be added to the FlowWorkList. If the value is $\mathbb{C}$, only the flow graph edge executed as the result of the branch is added to the FlowWorkList.[6]

3.4.1 *Asymptotic Complexity.* As in SSC, each SSA edge can only be examined twice. Nodes in the program flow graph are visited once for each of their in-edges. The asymptotic running complexity of this algorithm is the number of edges in the flow graph plus the number of SSA edges and should be linear in practice.

CC may be impractically slow and, consequently, was ignored for a long time. Many workers in code optimization had tried to derive practical sparse algorithms that achieved CC's results. However, they started from the sparse

[6]The value cannot be $\top$, since the earlier step in VisitExpression will have lowered the value.

A FRAGMENT OF SCC$_{ANF}$

$$
\begin{array}{llll}
\mathcal{A}[\![v]\!] & \Gamma & \Omega & = \langle \Gamma v,\, \Gamma,\, \Omega \rangle \\
\mathcal{A}[\![f(v_1, \ldots, v_n)]\!] & \Gamma & \Omega & \\
\quad \mid f \in \textbf{Prim} & & & = \mathcal{E}_{\textbf{Abs}}[\![f(\Gamma v_1, \ldots, \Gamma v_n)]\!] \\
\quad \mid \text{otherwise} & & & = \langle \Gamma' f,\, \Gamma',\, \textbf{if } \mathit{changed} \textbf{ then } \Omega \cup \{f\} \textbf{ else } \Omega \rangle
\end{array}
$$

$\textbf{where}$

f is defined as $f(x_1, \ldots, x_n) = e$

$\Gamma' = \Gamma \sqcap [f \mapsto \bot, x_1 \mapsto \Gamma v_1, \ldots, x_n \mapsto \Gamma v_n]$

$\mathit{changed} = \exists i. \Gamma x_i \sqsubset \Gamma v_i$ — indicates whether Γ changed

$\mathcal{A}[\![\texttt{letrec}\, f_1, \ldots, f_n \texttt{ in } e]\!]\Gamma \quad \Omega \quad =$

$\textbf{let}$

$\langle a, \Gamma', \Omega' \rangle = \mathcal{A}[\![e]\!]\Gamma\{\}$

$\langle \Gamma'', \Omega'' \rangle = \mathcal{A}_{\text{fix}}[\![f_1, \ldots, f_n]\!]\Gamma'(\Omega \cup \Omega')$

$\textbf{in}$

$\textbf{if } \Gamma' == \Gamma'' \textbf{ then } \langle a, \Gamma', \Omega'' \rangle \textbf{ else } \mathcal{A}[\![\texttt{letrec}\, f_1, \ldots, f_n \texttt{ in } e]\!]\Gamma''\Omega''$

$$
\begin{array}{ll}
\mathcal{A}_{\text{fix}}[\![\mathit{fun}_1, \ldots, \mathit{fun}_n]\!]\Gamma\Omega & \mid \nexists i. f_i \in \Omega = \langle \Gamma, \Omega \rangle \\
 & \mid \text{otherwise} \;=
\end{array}
$$

$\textbf{let}$

$\langle a, \Gamma', \Omega' \rangle = \mathcal{A}[\![e]\!]\Gamma\{\}$

$\Gamma'' = \Gamma' \sqcap [f_i \mapsto a]$

$\Omega'' = \Omega \cup \Omega' \setminus \{f_i\}$

$\textbf{in}$

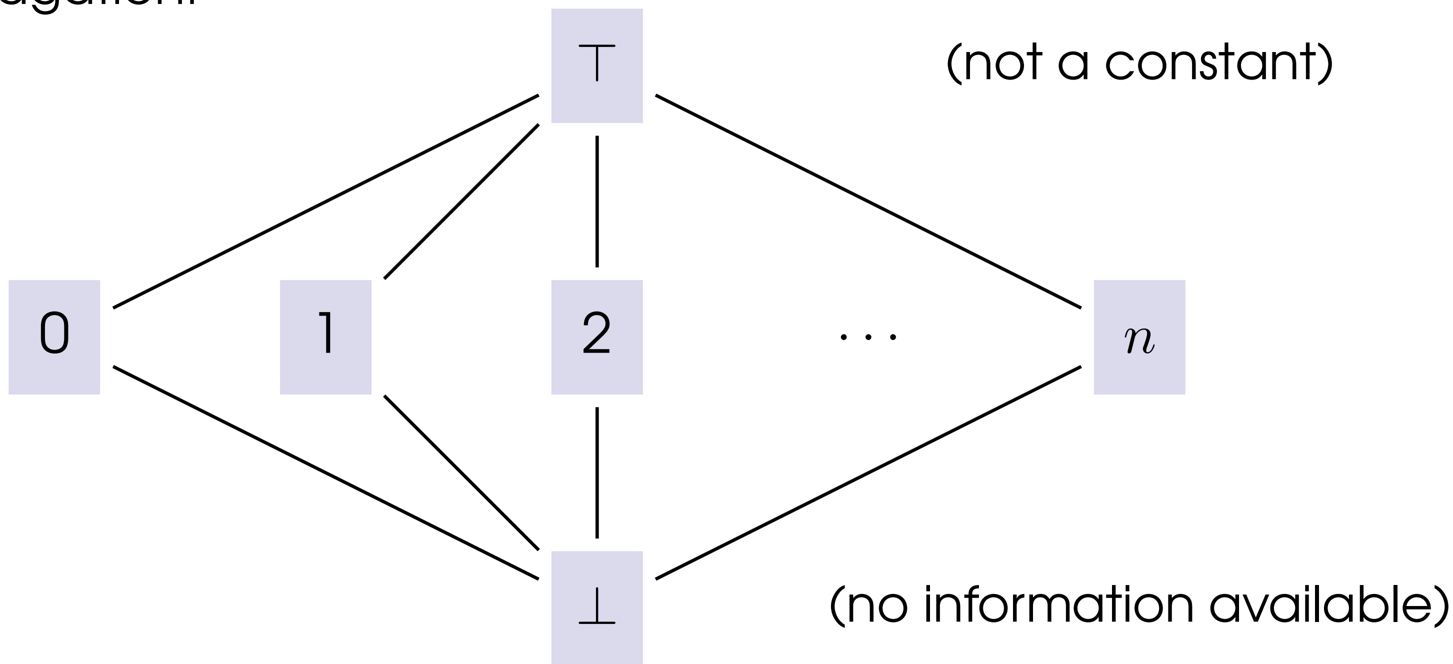
$\mathcal{A}_{\text{fix}}[\![\mathit{fun}_1, \ldots, \mathit{fun}_n]\!]\Gamma''(\textbf{if } \Gamma f_i \sqsubset \Gamma a \textbf{ then } \Omega'' \cup (\mathrm{Occ}\, f_i \cap \mathrm{Dom}\, \Gamma) \textbf{ else } \Omega'')$

$\textbf{where}$

$(f_i(x_1, \ldots, x_m) = e) = \mathit{fun}_i$

An overview of SCC$_{ANF}$

- ➜ Denoted in TEX-ised Haskell
- ➜ Operates on programs in ANF
- ➜ Split into "analysis" and "simplification" stages
- ➜ Operates over the standard lattice $\mathbb{N}^{\top}_{\perp}$ for constant propagation:



SO, HOW DOES IT WORK?

- ➜ Analysis proceeds by abstract interpretation under an environment Γ mapping variables to lattice values
- ➜ Maintains a work list Ω of functions to visit by
  - inspecting call sites and
  - tracking free variables for changes to abstract variables
- ➜ Functions added to the work list whenever any of its free variables is refined in the environment
- ➜ `letrec`'s are processed recursively until the work list is emptied
- ➜ The resulting abstract environment contains all information obtained about any constants
- ➜ Simplification phase replaces variables by constant values and eliminates redundant branches

```
letrec f(x) =


   let x’ = sub(x, x)


   in
     if x’ then
       f(x’)
     else
       x


in
   f(7)
```

```
letrec f(x) =


  let x' = sub(x, x)


  in
    if x' then
      f(x')
    else
      x


in
  f(7)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{f\}$$

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \perp, x \mapsto 7\}, \Omega = \{\}$$

```
   let x' = sub(x, x)


   in
     if x' then
       f(x')
     else
       x


in
   f(7)
```

$$\Gamma = \{f \mapsto \perp, x \mapsto 7\}, \Omega = \{f\}$$

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \perp, x \mapsto 7\}, \Omega = \{\}$$

```
   let x' = sub(x, x)
```

$$\Gamma = \{f \mapsto \perp, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
     else
       x

in
   f(7)
```

$$\Gamma = \{f \mapsto \perp, x \mapsto 7\}, \Omega = \{f\}$$

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$

```
   let x' = sub(x, x)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
     else
       x
```

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{f\}$$

```
in
   f(7)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{f\}$$

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$

```
   let x' = sub(x, x)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
     else
       x
```

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{f\}$$

```
in
   f(7)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{f\}$$

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{f\}$$

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

```
   let x' = sub(x, x)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
     else
       x
```

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{f\}$$

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{\}$$

```
in
   f(7)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{f\}$$

$$\Gamma = \{f \mapsto 7, x \mapsto 7, x' \mapsto 0\}, \Omega = \{f\}$$

```
letrec f(x) =


   let x’ = sub(x, 1)


   in
     if x’ then
       f(x’)


     else
       x

in
   f(7)
```

```
letrec f(x) =

  let x' = sub(x, 1)

  in
    if x' then
      f(x')

    else
      x

in
  f(7)
```

$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$

```
letrec f(x) =
```

$\Gamma = \{f \mapsto \perp, x \mapsto 7\}, \Omega = \{\}$

```
  let x' = sub(x, 1)
```

$\Gamma = \{f \mapsto \perp, x \mapsto 7, x' \mapsto 6\}, \Omega = \{\}$

```
  in
    if x' then
      f(x')

    else
      x

in
  f(7)
```

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$

```
   let x' = sub(x, 1)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 6\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
```

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{f\}$$

```
     else
       x
in
   f(7)
```

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{\}$$

```
   let x' = sub(x, 1)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 6\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
```

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{f\}$$

```
     else
       x

in
   f(7)
```

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$
$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{\}$$

```
  let x' = sub(x, 1)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 6\}, \Omega = \{\}$$
$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto \top\}, \Omega = \{\}$$

```
  in
    if x' then
      f(x')
```

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{f\}$$

```
    else
      x

in
  f(7)
```

```
letrec f(x) =
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7\}, \Omega = \{\}$$

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{\}$$

```
   let x' = sub(x, 1)
```

$$\Gamma = \{f \mapsto \bot, x \mapsto 7, x' \mapsto 6\}, \Omega = \{\}$$

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto \top\}, \Omega = \{\}$$

```
   in
     if x' then
       f(x')
```

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto 6\}, \Omega = \{f\}$$

$$\Gamma = \{f \mapsto \bot, x \mapsto \top, x' \mapsto \top\}, \Omega = \{\}$$

```
     else
       x
```

$$\Gamma = \{f \mapsto \top, x \mapsto \top, x' \mapsto \top\}, \Omega = \{f\}$$

```
in
   f(7)
```

Work complexity

① Each variable updated at most three times
(same as SCC_{SSA})

② Each function processed once for each change to its free variables
(in SCC_{SSA}, processed once for each incoming SSA edge)

③ Number of free variables corresponds directly to number of SSA edges

④ Therefore, SCC_{ANF} has the same work comlexity as SCC_{SSA}.

OBSERVATIONS

① SCC$_{ANF}$ performs inter-procedural analysis transparently, although alias analysis and performance still a problem

② Vanilla SCC$_{ANF}$ cannot handle higher-order functions

③ The algorithm does not preserve non-termination

④ Optimistic algorithms are difficult to analyze:

- environment invalid until fixpoint reached
- very complex invariants

Conclusions

Benefits of ANF:

- ➜ Simplifies the operational semantics, the environment and therefore static analysis and formal reasoning
- ➜ Allows us to view imperative programs as encodings of functional programs
- ➜ Allows us to adopt SSA algorithms for compilation of functional programs
- ➜ Integrates intra- and inter-procedural analysis
- ➜ Straight forward extension to higher-order functions