
A Functional Perspective on SSA Optimization Algorithms

Patryk Zadarnowski

jointly with

Manuel M. T. Chakravarty

Gabriele Keller

19 April 2004

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
- People tried and failed in the past (IBM Vienna project)
- Functional programmers eat and breathe formal methods!

THE APPROACH

- ① Fix the problem, not the tools:

If compilers are hard to verify,
change them to make it easier!

- ② Fix data structures before fixing algorithms:

In every FORTRAN program hides
a functional program
trying to break free!

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

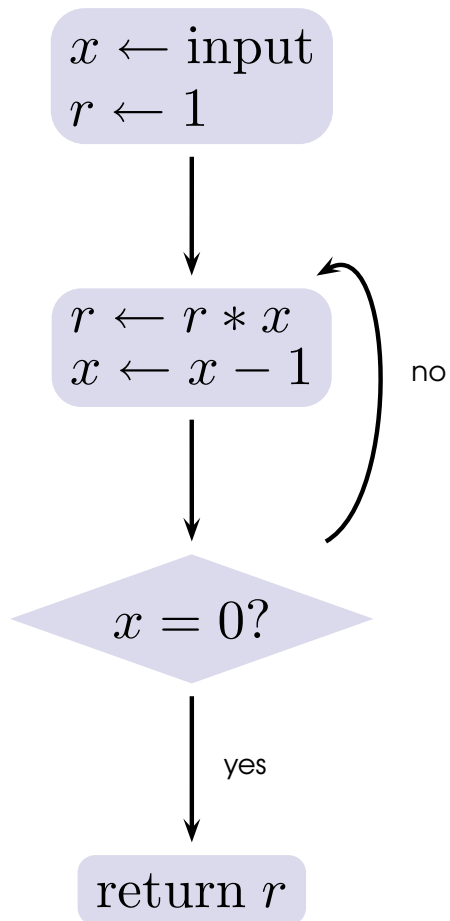
Also:

- ④ Formal operational semantics for the SSA form.
- ⑤ A new flexible data-flow framework.
- ⑥ Solves the multiple- ϕ problem of the SSA form (Robert Morgan.)
- ⑦ Received very enthusiastic response from both imperative and functional compiler communities.

THE PLAN

- ① Start with the Static Single Assignment (SSA) Form
 - Appel showed informally that SSA is in fact a variant of lambda calculus
 - Kelsey demonstrated correspondence between SSA and the continuation-passing style (CPS) of lambda calculus
 - However, Flanagan *et al.* showed that CPS is bad for data flow analysis.
- ② Design an algorithm for translating SSA programs to the Administrative Normal Form (ANF) of lambda calculus.
- ③ Use the correspondence introduced by the translation function to implement some traditional data-flow algorithms on ANF.
- ④ Use formal semantics of ANF to prove properties about those algorithm.

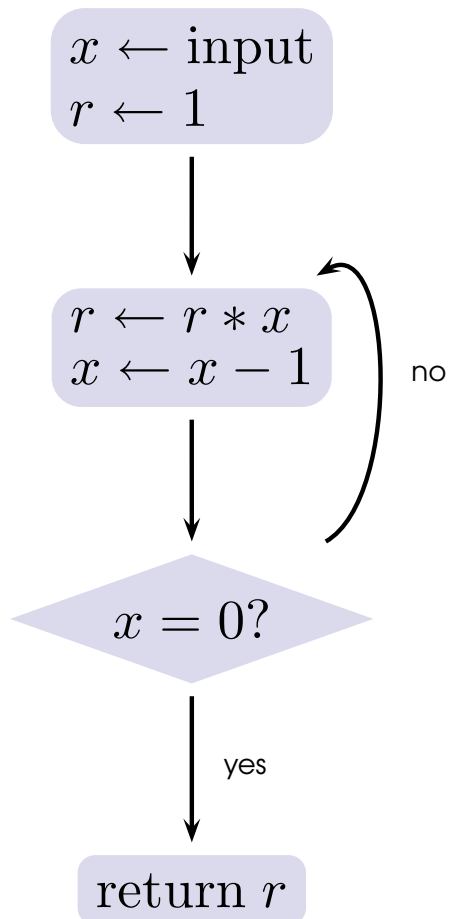
CONTROL FLOW GRAPHS



Summary:

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

CONTROL FLOW GRAPHS



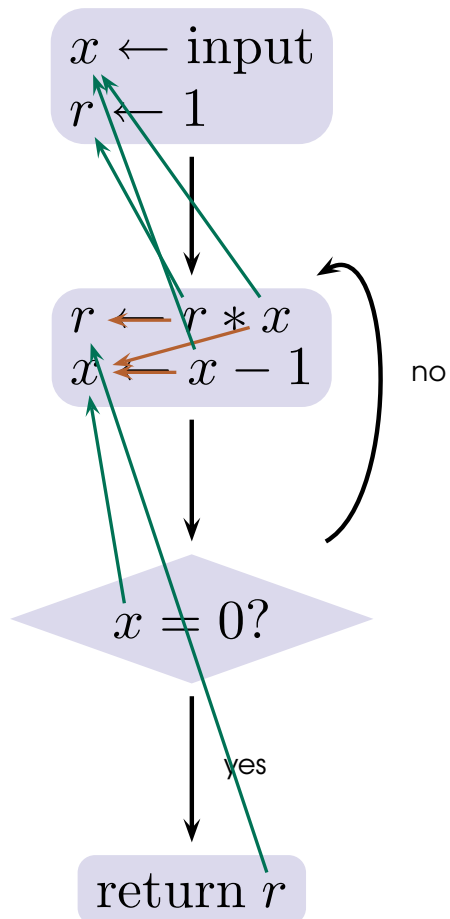
Summary:

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

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

- Expensive to compute
- Reusable

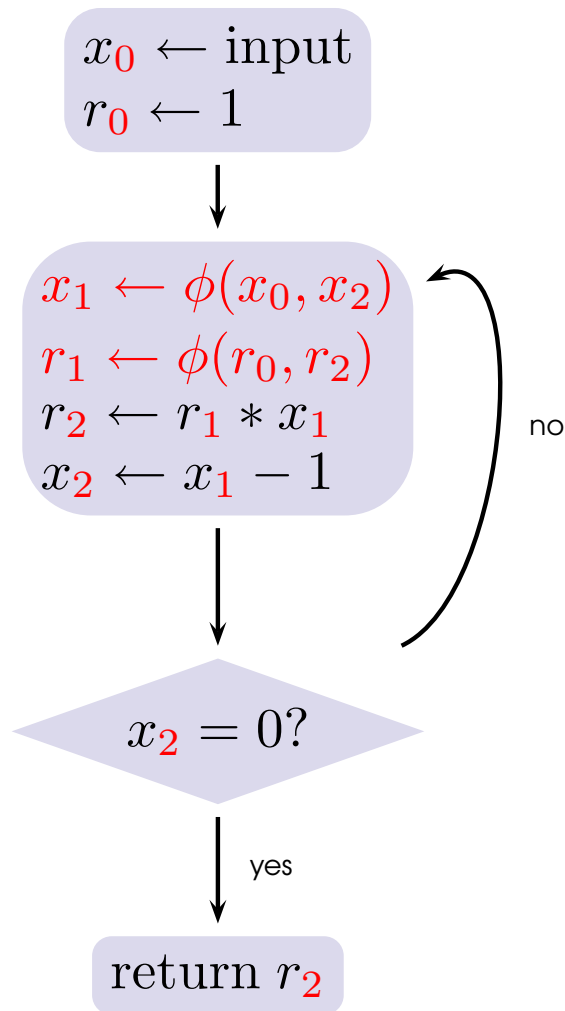
CONTROL FLOW GRAPHS & DATAFLOW INFORMATION



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 occurrence
- Values from different control flows merged via ϕ functions
- ϕ functions placed by computing the *dominator tree* of the procedure

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$

Syntax:

- procedures

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$
 $b ::= e \mid b; x:e \mid$
 $b_1; x:\{b_2\}$

Syntax:

- procedures
- basic blocks

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$
 $b ::= e \mid b; x:e \mid$
 $\quad b_1; x:\{b_2\}$
 $e ::= x \leftarrow v; e$

Syntax:

- procedures
- basic blocks
- copy

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$
 $b ::= e \mid b; x:e \mid$
 $b_1; x:\{b_2\}$
 $e ::= x \leftarrow v; e \mid$
 $x \leftarrow v(\bar{v}); e$

Syntax:

- procedures
- basic blocks
- copy
- calls

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$
 $b ::= e \mid b; x:e \mid$
 $b_1; x:\{b_2\}$
 $e ::= x \leftarrow v; e \mid$
 $x \leftarrow v(\bar{v}); e \mid$
 $\text{goto } x;$

Syntax:

- procedures
- basic blocks
- copy
- calls
- jumps

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$
 $b ::= e \mid b; x:e \mid$
 $b_1; x:\{b_2\}$
 $e ::= x \leftarrow v; e \mid$
 $x \leftarrow v(\bar{v}); e \mid$
 $\text{goto } x; \mid$
 $\text{ret } v; \mid \text{ret } v(\bar{v});$

Syntax:

- procedures
- basic blocks
- copy
- calls
- jumps
- returns

STATIC SINGLE ASSIGNMENT FORM — DETAILS

$p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$
 $b ::= e \mid b; x:e \mid$
 $b_1; x:\{b_2\}$
 $e ::= x \leftarrow v; e \mid$
 $x \leftarrow v(\bar{v}); e \mid$
 $\text{goto } x; \mid$
 $\text{ret } v; \mid \text{ret } v(\bar{v}); \mid$
 $\text{if } v \text{ then } e_1 \text{ else } e_2$

Syntax:

- procedures
- basic blocks
- copy
- calls
- jumps
- returns
- branches

STATIC SINGLE ASSIGNMENT FORM — DETAILS

```
 $p ::= \text{proc } x(\bar{x}) \{b\} p \mid e$   
 $b ::= e \mid b; x:e \mid$   
     $b_1; x:\{b_2\}$   
 $e ::= x \leftarrow v; e \mid$   
     $x \leftarrow v(\bar{v}); e \mid$   
     $\text{goto } x; \mid$   
     $\text{ret } v; \mid \text{ret } v(\bar{v}); \mid$   
     $\text{if } v \text{ then } e_1 \text{ else } e_2 \mid$   
     $x \leftarrow \phi(\bar{g}); e$   
 $g ::= l:v$   
 $l ::= x \mid \text{start}$ 
```

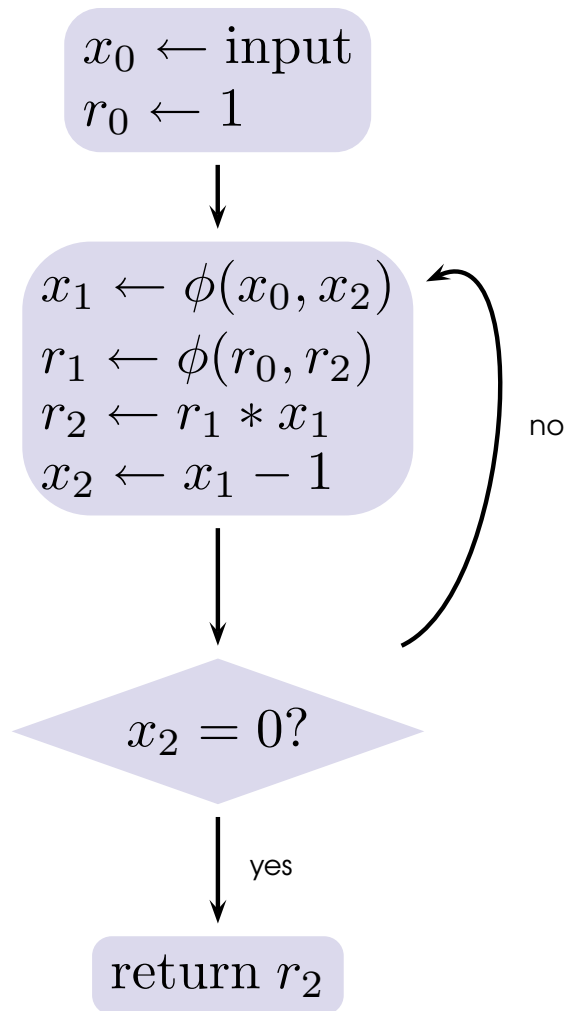
Syntax:

- procedures
- basic blocks
- copy
- calls
- jumps
- returns
- branches
- phi nodes

STRUCTURED SSA FORM

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

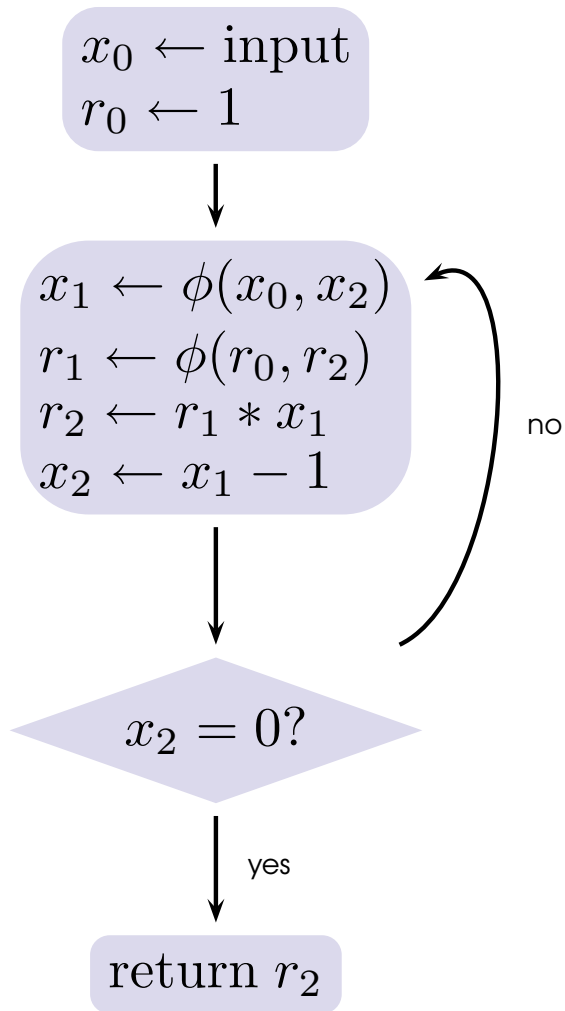
STATIC SINGLE ASSIGNMENT FORM



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
 - ...

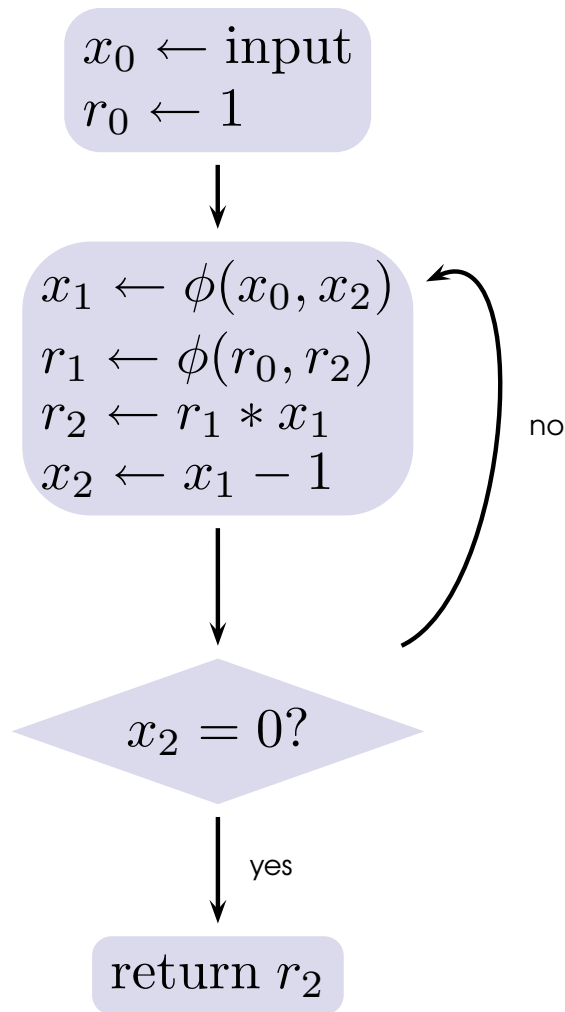
STATIC SINGLE ASSIGNMENT FORM



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

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

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

Syntax:

- copy

ADMINISTRATIVE NORMAL FORM

$e ::= \text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(\bar{v}) \text{ in } e$

Syntax:

- copy
- calls

ADMINISTRATIVE NORMAL FORM

$e ::= \text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(\bar{v}) \text{ in } e \mid$
 v

Syntax:

- copy
- calls
- returns

ADMINISTRATIVE NORMAL FORM

$e ::= \text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(\bar{v}) \text{ in } e \mid$
 $v \mid$
 $v(\bar{v})$

Syntax:

- copy
- calls
- returns
- jumps

ADMINISTRATIVE NORMAL FORM

$e ::= \text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(\bar{v}) \text{ in } e \mid$
 $v \mid$
 $v(\bar{v}) \mid$
 $\text{if } v \text{ then } e_1 \text{ else } e_2$

Syntax:

- copy
- calls
- returns
- jumps
- branches

ADMINISTRATIVE NORMAL FORM

Syntax:

$e ::= \text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(\bar{v}) \text{ in } e \mid$
 $v \mid$
 $v(\bar{v}) \mid$
 $\text{if } v \text{ then } e_1 \text{ else } e_2 \mid$
 $\text{letrec } \bar{f} \text{ in } e$
 $f ::= x(\bar{x}) = e$

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

ADMINISTRATIVE NORMAL FORM — EXAMPLE

```
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  
  ...
```

ADMINISTRATIVE NORMAL FORM

$e ::= \text{let } x = v \text{ in } e \mid$
 $\text{let } x = v(\bar{v}) \text{ in } e \mid$
 $v \mid$
 $v(\bar{v}) \mid$
 $\text{if } v \text{ then } e_1 \text{ else } e_2 \mid$
 $\text{letrec } \bar{f} \text{ in } e$
 $f ::= x(\bar{x}) = e$

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)

TRANSLATING STRUCTURED SSA TO ANF

```
proc fac(x) {
  L1: {
    L2:
      in
      in
    }
  }
  ...
}
```

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

TRANSLATING STRUCTURED SSA TO ANF

```
proc fac(x) {
  L1: {
    if x0 < 0 then else ret r0
    L2: r1 ← mul(r0, x0)
       x1 ← sub(x0, 1)
  }
}
...
```

```
letrec fac(x) =
  letrec fac'(x) =
    letrec fac''(x) =
      let r1 = mul(r0, x0)
      let x1 = sub(x0, 1)
    in
      if x0 < 0 then 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
    ...
```

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 ←  $\phi$ (start:1, L1:x1)  
    x1 ← sub(x0, 1)  
    if x1 then  
        goto L1  
    else  
        ret 7  
}
```

→

```
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 i th in-edge, the result of the ϕ -function is the value of its i th 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 τ .

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 τ .
- (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 τ .

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 \perp , all exit edges must be added to the *FlowWorkList*. If the value is \emptyset , 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

⁶The value cannot be τ , since the earlier step in *VisitExpression* will have lowered the value.

A FRAGMENT OF SCC_{ANF}

$$\begin{aligned}
\mathcal{A}[[v]] \quad \Gamma \quad \Omega &= \langle \Gamma v, \Gamma, \Omega \rangle \\
\mathcal{A}[[f(v_1, \dots, v_n)]] \quad \Gamma \quad \Omega &= \mathcal{E}_{\mathbf{Abs}}[[f(\Gamma v_1, \dots, \Gamma v_n)]] \\
&| f \in \mathbf{Prim} \\
&| \text{otherwise} &= \langle \Gamma' f, \Gamma', \mathbf{if\ changed\ then\ } \Omega \cup \{f\} \mathbf{\ else\ } \Omega \rangle
\end{aligned}$$

where

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

$$\begin{aligned}
\Gamma' &= \Gamma \sqcap [f \mapsto \perp, x_1 \mapsto \Gamma v_1, \dots, x_n \mapsto \Gamma v_n] \\
\text{changed} &= \exists i. \Gamma x_i \sqsubset \Gamma v_i \quad \text{-- indicates whether } \Gamma \text{ changed}
\end{aligned}$$

$$\mathcal{A}[[\text{letrec } f_1, \dots, f_n \text{ in } e]] \Gamma \quad \Omega =$$

let

$$\begin{aligned}
\langle a, \Gamma', \Omega' \rangle &= \mathcal{A}[[e]] \Gamma \{ \} \\
\langle \Gamma'', \Omega'' \rangle &= \mathcal{A}_{\mathbf{fix}}[[f_1, \dots, f_n]] \Gamma' (\Omega \cup \Omega')
\end{aligned}$$

in

$$\mathbf{if\ } \Gamma' = \Gamma'' \mathbf{\ then\ } \langle a, \Gamma', \Omega'' \rangle \mathbf{\ else\ } \mathcal{A}[[\text{letrec } f_1, \dots, f_n \text{ in } e]] \Gamma'' \Omega''$$

$$\mathcal{A}_{\mathbf{fix}}[[fun_1, \dots, fun_n]] \Gamma \Omega \quad | \nexists i. fun_i \in \Omega = \langle \Gamma, \Omega \rangle$$

| otherwise =

let

$$\begin{aligned}
\langle a, \Gamma', \Omega' \rangle &= \mathcal{A}[[e]] \Gamma \{ \} \\
\Gamma'' &= \Gamma' \sqcap [fun_i \mapsto a] \\
\Omega'' &= \Omega \cup \Omega' \setminus \{fun_i\}
\end{aligned}$$

in

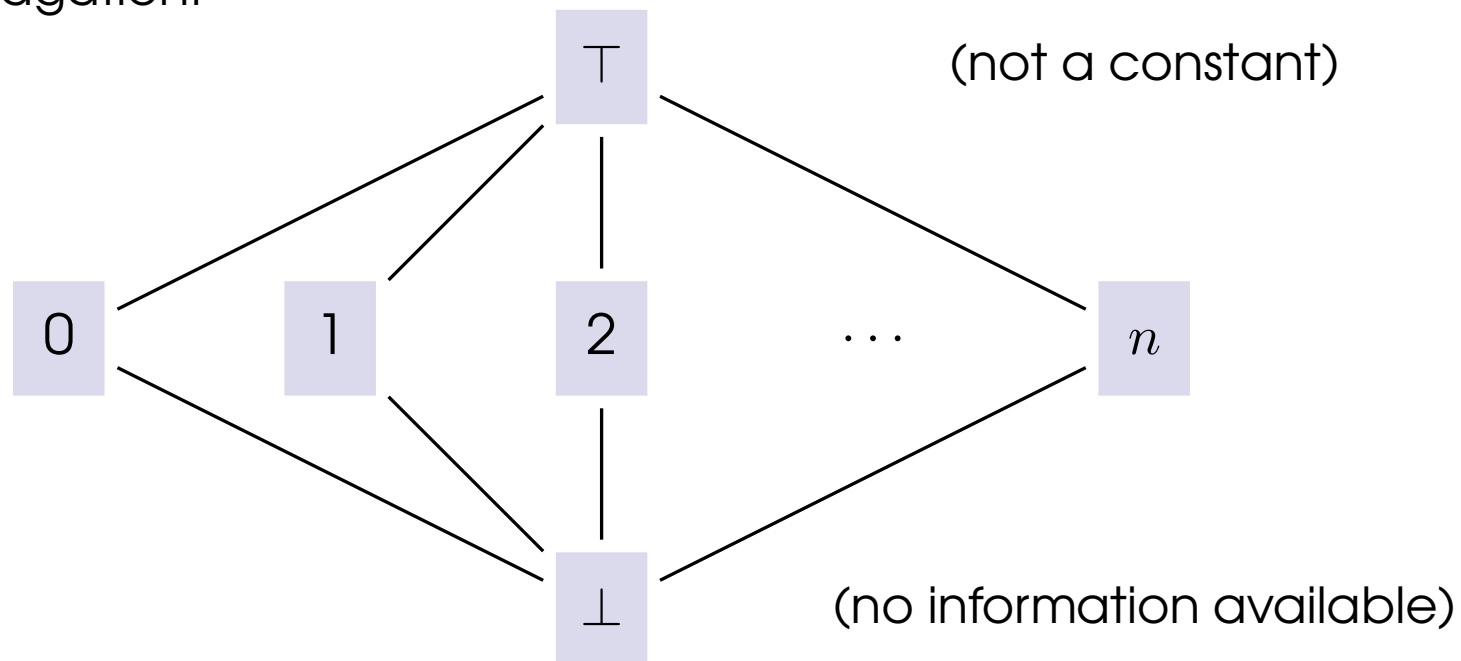
$$\mathcal{A}_{\mathbf{fix}}[[fun_1, \dots, fun_n]] \Gamma'' (\mathbf{if\ } \Gamma fun_i \sqsubset \Gamma a \mathbf{\ then\ } \Omega'' \cup (\text{Occ } fun_i \cap \text{Dom } \Gamma) \mathbf{\ else\ } \Omega'')$$

where

$$(fun_i(x_1, \dots, x_m) = e) = fun_i$$

AN OVERVIEW OF SCC_{ANF}

- Denoted in $\text{T}_{\text{E}}\text{X}$ -ised Haskell
- Operates on programs in ANF
- Split into “analysis” and “simplification” stages
- Operates over the standard lattice $\mathbb{N}_{\perp}^{\top}$ 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)

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

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) =

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

let x' = sub(x, x)

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

in

if x' then

f(x')

else

x

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

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

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

in

f(7)

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

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

letrec f(x) =

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

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

let x' = sub(x, x)

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

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

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)

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

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

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

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

```
  in
```

```
    if x' then
```

```
      f(x')
```

```
    else
```

```
      x
```

```
in
```

```
  f(7)
```

letrec f(x) =

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

let x' = sub(x, 1)

in

if x' then

f(x')

else

x

in

f(7)

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) =

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

let x' = sub(x, 1)

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

in

if x' then

f(x')

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

else

x

in

f(7)

letrec f(x) =

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

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

let x' = sub(x, 1)

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

in

if x' then

f(x')

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

else

x

in

f(7)

letrec f(x) =

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

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

let x' = sub(x, 1)

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

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

in

if x' then

f(x')

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

else

x

in

f(7)

letrec f(x) =

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

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

let x' = sub(x, 1)

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

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

in

if x' then

f(x')

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

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

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 complexity 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

THE PAPER

<http://www.cse.unsw.edu.au/~patrykz/papers/ssa-lambda/>

THE ANALYSIS FUNCTION: VALUES

$$\mathcal{A}[[v]]\Gamma \Omega = \langle \Gamma v, \Gamma, \Omega \rangle$$

THE ANALYSIS FUNCTION: VALUES & APPLICATIONS

$$\mathcal{A}[[v]]\Gamma \Omega = \langle \Gamma v, \Gamma, \Omega \rangle$$

$$\mathcal{A}[[f(v_1, \dots, v_n)]]\Gamma \Omega$$

$$| f \in \text{Prim} \quad = \mathcal{E}_{\text{Abs}}[[f(\Gamma v_1, \dots, \Gamma v_n)]]$$

$$| \text{otherwise} \quad = \langle \Gamma' f, \Gamma', \text{if changed then } \Omega \cup \{f\} \text{ else } \Omega \rangle$$

where

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

$$\Gamma' = \Gamma \sqcap [f \mapsto \perp, x_1 \mapsto \Gamma v_1, \dots, x_n \mapsto \Gamma v_n]$$

$$\text{changed} = \exists i. \Gamma x_i \sqsubset \Gamma v_i \quad \text{-- indicates whether } \Gamma \text{ changed}$$

THE ANALYSIS FUNCTION: `let`

$\mathcal{A}[\text{let } x=e_1 \text{ in } e_2]\Gamma \Omega = \mathbf{let}$

$$\langle a, \Gamma', \Omega' \rangle = \mathcal{A}[e_1]\Gamma \Omega$$

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

$$\text{changed} = \Gamma x \sqsubset \Gamma a$$

$$\text{affected} = \text{Occ } x \cap \text{Dom } \Gamma$$

-- $\cap \text{Dom } \Gamma$ removes not yet used functions

in

$\mathcal{A}[e_2]\Gamma''$ (**if** *changed* **then** $\Omega \cup \text{affected}$ **else** Ω)

THE ANALYSIS FUNCTION: if-then

$$\begin{aligned} \mathcal{A}[\text{if } v \text{ then } e_1 \text{ else } e_2] \Gamma \Omega & \\ | \Gamma v == \perp & = \langle \perp, \Gamma, \Omega \rangle \\ | \Gamma v == \text{True} & = \mathcal{A}[e_1] \Gamma \Omega \\ | \Gamma v == \text{False} & = \mathcal{A}[e_2] \Gamma \Omega \\ | \Gamma v == \top & = \mathbf{let} \\ & \quad \langle a_1, \Gamma_1, \Omega_1 \rangle = \mathcal{A}[e_1] \Gamma \Omega \\ & \quad \langle a_2, \Gamma_2, \Omega_2 \rangle = \mathcal{A}[e_2] \Gamma_1 \Omega_1 \\ & \quad \mathbf{in} \\ & \quad \langle a_1 \sqcap a_2, \Gamma_2, \Omega_2 \rangle \end{aligned}$$

THE ANALYSIS FUNCTION: letrec

$$\mathcal{A}[\text{letrec } f_1, \dots, f_n \text{ in } e] \Gamma \quad \Omega \quad =$$

let

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

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

in

$$\begin{aligned} & \text{if } \Gamma' == \Gamma'' \text{ then } \langle a, \Gamma', \Omega'' \rangle \text{ else } \mathcal{A}[\text{letrec } f_1, \dots, f_n \text{ in } e] \Gamma'' \Omega'' \\ & \mathcal{A}_{\text{fix}}[\text{fun}_1, \dots, \text{fun}_n] \Gamma \Omega \mid \nexists i. f_i \in \Omega = \langle \Gamma, \Omega \rangle \\ & \quad \mid \text{otherwise} = \end{aligned}$$

let

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

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

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

in

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

where

$$(f_i(x_1, \dots, x_m) = e) = fun_i$$