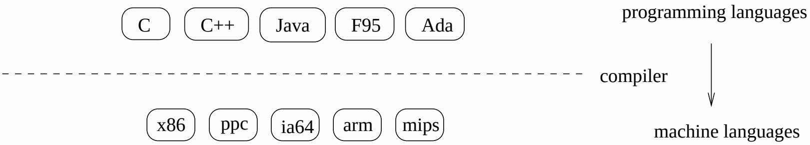
The SSA Representation Framework: Semantics, Analyses and GCC Implementation

Sebastian Pop

CRI, École des mines de Paris, France

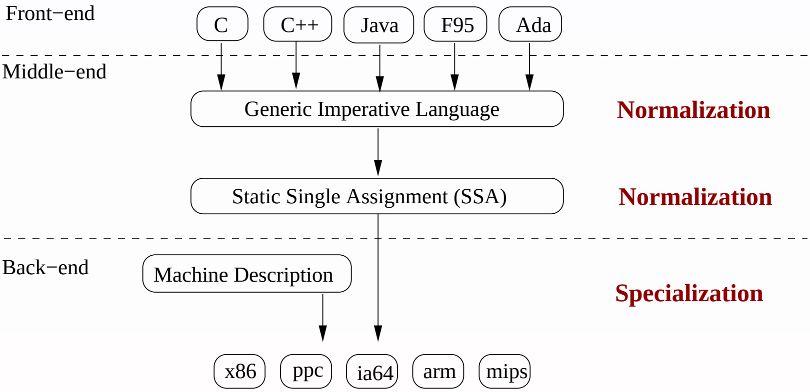
PhD Defense,
Paris, December 13, 2006

Introduction: languages machines and compilers



compilers are translators between languages

Structure of Modern Compilers



SSA used for reducing the complexity of static scalar analyses

SSA Representation

```
a = 1
do forever
  a = a + 3
```

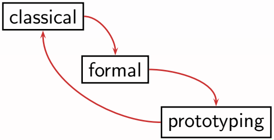
SSA representation ⟶

```
a = 1
loop
  b = phi (a, c)
  c = b + 3
endloop
```

- use-def links,
- phi nodes at control flow junctions.

Overview

1. an algorithm on classical SSA: scalar evolutions analysis
2. formal SSA framework
3. natural description of SSA algorithms in declarative languages



Part 1: Loop based SSA and evolutions of scalar variables

Induction Variables (IV)

```
for i = 0 to N
  a = ...
```

- variable `a` is an **induction variable**: its value may change with successive `i` values.
- goal: describe the values taken by scalar variables in loops
  - give the successive values (when possible),
  - give a range or an envelope of values.

Chains of Recurrences

- representation of successive values in loops using a form called **multivariate chains of recurrences (MCR)**.
- for instance, the chain of recurrence

$$\{1, +, 3\}$$

represents the evolution of scalar variable "a" in the program:

```
a = 1
do forever
  a = a + 3
```

Induction Variable Analysis

Algorithm:

1. Walk the use-def edges, find a SCC, (Tarjan algorithm with backtrack)
2. Reconstruct the update expression,
3. Translate to a chain of recurrence,
4. (optional) Instantiate parameters.

Example: finding the evolution of scalar "c"

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

```
a = 3;
b = 1;
while (b ≤ 123) do
  a = a + b + 7
  b = b + 5
```

Depth-first walk the use-defs to a loop-phi node: $c \rightarrow f \rightarrow e \rightarrow d$
$d \neq c$, backtrack

Example: finding the evolution of scalar “c”

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

Found the starting loop-phi. The SCC is:

$$c \longrightarrow f \longrightarrow c$$

Example: finding the evolution of scalar “c”

```
a = 3
b = 1
loop
   c = phi (a, f)
   d = phi (b, g)
   if (d > 123) goto end
   e = d + 7
   f = e + c
   g = d + 5
endloop
end:
```

Reconstruct the update expression: $c + e$

$$c = phi\ (a,\ c + e) \rightarrow \{a, +, e\}$$

Example: finding the evolution of scalar “c”

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \longrightarrow \{a, +, e\} \xrightarrow{Instantiate} Optional \cdots$$

Example: finding the evolution of scalar "c"

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

Example: finding the evolution of scalar “c”

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

$$e \rightarrow d + 7$$

Example: finding the evolution of scalar "c"

```
a = 3
b = 1
loop
   c = phi (a, f)
   d = phi (b, g)
   if (d > 123) goto end
   e = d + 7
   f = e + c
   g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

$$e \rightarrow d + 7$$

$$d \rightarrow \{1, +, 5\}$$

Example: finding the evolution of scalar "c"

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

$$e \rightarrow \{8, +, 5\}$$

$$d \rightarrow \{1, +, 5\}$$

Example: finding the evolution of scalar “c”

```
a = 3
b = 1
loop
   c = phi (a, f)
   d = phi (b, g)
   if (d > 123) goto end
   e = d + 7
   f = e + c
   g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, 8, +, 5\}(x) = 3\binom{x}{0} + 8\binom{x}{1} + 5\binom{x}{2}$$

$$e \rightarrow \{8, +, 5\}$$

Applications

```
b = 1
loop
   d = phi (b, g)
   if (d > 123) goto end
   g = d + 5
endloop
end:
h = phi (d)
```

$\xrightarrow{scevCP}$

```
b = 1
loop
   d = phi (b, g)
   if (d > 123) goto end
   g = d + 5
endloop
end:
h = 126
```

- computing the number of iterations in a loop
- constant propagation after loops

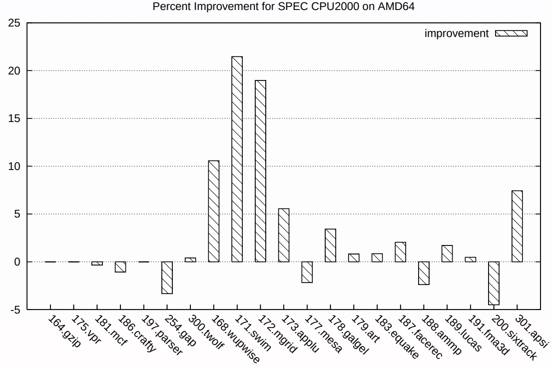
Analysis of scalar evolutions (scev) in GCC

- SSA → MCR implemented in the GNU Compiler Collection
- scev is fast and stable: 2 years in production GCC (4.x)

other components based on scev

- data dependence analysis (Banerjee, gcd, etc.)
- unimodular transformations of loop nests (interchange)
- vectorization
- scalar variable optimizations
- value range propagation
- parallelization

Experiments: CPU2000 on AMD64 3700 Linux 2.6.13



- GCC version 4.1 as of 2005-Nov-04
- options: “-O3 -msse2 -ftree-vectorize -ftree-loop-linear”
- base: scev analyzer disabled

Part 2: Formal framework for SSA

Formal framework for SSA

This was a classical presentation of an algorithm working on SSA

- description in natural language
- informal definitions: semantics by examples
- enough information for engineering a similar analyzer

However

- imprecise description of algorithms
- impossible to prove correctness
- impractical graphical representation
- impossible to use classical abstract interpretation

Syntax of SSA

SSA expressions are defined as follows:

$$
\begin{array}{rcl}
N & \in & Cst \\
I & \in & Ide \\
E & \in & SSA ::= N \mid I \mid E_1 \oplus E_2 \mid \mathrm{loop}_\ell\phi(E_1, E_2) \mid \mathrm{close}_\ell\phi(E_1, E_2)
\end{array}
$$

Denotational semantics of SSA

- associate an expression to each SSA identifier, σ: I$\longrightarrow$E
- iteration vectors, $k \in N^m$

$$
\begin{aligned}
\mathcal{E}[\![\mathrm{loop}_\ell\phi(E_1, E_2)]\!]\sigma k &= \begin{cases} \mathcal{E}[\![E_1]\!]\sigma k, & \text{if } k_\ell = 0, \\ \mathcal{E}[\![E_2]\!]\sigma k_{\ell-}, & \text{otherwise.} \end{cases} \\
\mathcal{E}[\![\mathrm{close}_\ell\phi(E_1, E_2)]\!]\sigma k &= \mathcal{E}[\![E_2]\!]\sigma k[\min\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}/\ell]
\end{aligned}
$$

- loopϕ provides values for some k (primitive recursive)
- closeϕ contains a minimization operator (partial recursive)

Discussion

- there is no assignment in the “Static Single Assignment” form!
- SSA is a declarative language
- semantics of SSA based on partial recursive functions
- minimization operator intrinsic to SSA language

Recording semantics of Imp (intuitive idea)

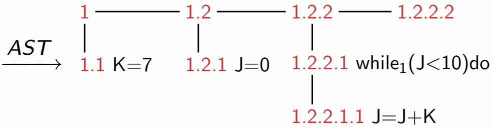
Imp (a simple imperative language) is defined by:

$$
\begin{array}{rcl}
N & \in & Cst \\
I & \in & Ide \\
E & \in & Expr ::= N \mid I \mid E_1 \oplus E_2 \\
S & \in & Stmt ::= I = E \mid S_1; S_2 \mid \text{while}_\ell\ E \text{ do } S
\end{array}
$$

- loops uniquely identified by ℓ
- record at each point of the program every computed value
- point = static + dynamic information
    - static = text sequence order (h)
    - dynamic = loop iteration order (k)

Identifying statements in Imp: static, dynamic points

K = 7;
J = 0;
while_1(J < 10) do
 J = J + K;



K = 7;
J = 0;
while_{ℓ_1}(J<10)do
 J=J+K;
while_{ℓ_2}(J<100)do
 J=J+K;

$$\xrightarrow{\textit{exec order}} \begin{pmatrix}\ell_1\\ \ell_2\end{pmatrix} : \begin{pmatrix}0\\0\end{pmatrix} \prec \begin{pmatrix}1\\0\end{pmatrix} \prec \begin{pmatrix}2\\0\end{pmatrix} \prec \begin{pmatrix}2\\1\end{pmatrix} \prec \ldots \prec \begin{pmatrix}2\\13\end{pmatrix}$$

Consistency of translation

$$
\begin{array}{ccc}
\text{Imp} & \xrightarrow{\mathcal{C}[\![]\!] h \mu} & \text{SSA} \\
\mathcal{I}[\![]\!](h,k)t \downarrow & & \downarrow \mathcal{E}[\![]\!]\sigma k \\
v \in \mathcal{V} & \Longleftrightarrow\!= & v \in \mathcal{V}
\end{array}
$$

consistency property holds after translating any Imp stmt

Part 3: Abstract SSA in PROLOG

SSA in PROLOG

$$\begin{array}{l} K = 7; \\ J = 0; \\ \text{while}_1\ (J < 10)\ \text{do} \\ \quad J = J + K; \end{array} \xrightarrow{\text{PROLOG}}$$

```
ssa(K, 7).
ssa(J, 0).
ssa(A, lphi(l1, J, B)).
ssa(B, A + K).
ssa(C, cphi(l1, A < 10, A)).
```

SSA declarations represented by PROLOG facts

From SSA to chains of recurrences

```
fromSSAtoMCR(ssa(X,lphi(_,_,X+Step)), ssa(X,unknown)) :-
  hasAself(X, Step) .
fromSSAtoMCR(ssa(X, lphi(LoopId, Init, X + Step)),
             ssa(X, mcr(LoopId, Init, Step))).

hasAself(X, X).
hasAself(X, Name, Step) :- ssa(Name, Expr), hasAself(X, Expr).
hasAself(X, A + B) :- hasAself(X, A); hasAself(X, B).
```

- some information is lost (masking abstraction)
- use the unification engine of PROLOG(backtrack)

Discussion

- PROLOG is a natural language for representing SSA
- unification is used in classical algorithms on SSA
- scalar evolution algorithm simpler to describe in PROLOG

Conclusion

- theoretical framework for SSA
  - SSA is a declarative language (no use of imperative constructs)
  - paves the way to formal proofs for compiler correctness
  - allows the application of abstract interpretation framework
  - alternative proof for Turing's Equivalence Theorem
- prototyping framework in PROLOG
  - simple way to prototype SSA transformations
  - simplifies specification of algorithms on SSA
- practical implementations of static analyzers
  - implementations are stable and fast
  - integrated in an industrial compiler
  - in production for two years in GCC versions 4.x

Publications: conferences, workshops, and research reports

- Denotational Semantics for SSA Conversion. Sebastian Pop, Albert Cohen, Pierre Jouvelot, Georges-André Silber. Research report, June 2006.
- GRAPHITE: Polyhedral Analyses and Optimizations for GCC. Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, Nicolas Vasilache. GCC Summit 2006, Ottawa, Canada.
- The New Framework for Loop Nest Optimizations in GCC: from Prototyping to Evaluation. Sebastian Pop, Albert Cohen, Pierre Jouvelot, Georges-André Silber. The 12th Workshop on Compilers for Parallel Computers, CPC2006, January 2006, A Coruña, Spain.
- Induction Variable Analysis with Delayed Abstractions. Sebastian Pop, Albert Cohen, Georges-André Silber. First International Conference, High Performance Embedded Architectures and Compilers, HiPEAC2005, November 2005, Barcelona, Spain.
- High-Level Loop Optimizations for GCC. Daniel Berlin, David Edelsohn (IBM T.J. Watson Research Center), Sebastian Pop. GCC Summit 2004, Ottawa, Canada.
- Fast Recognition of Scalar Evolutions on Three-Address SSA Code. Sebastian Pop, Philippe Clauss (ICPS-LSIIT), Albert Cohen (INRIA), Vincent Loechner (ICPS-LSIIT), Georges-André Silber. Research report, October 2004.

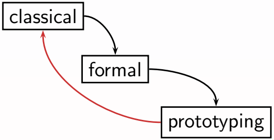
Future work: out of SSA

$$\begin{array}{ccccc}
\text{Imp} & \xrightarrow{\mathcal{C}[\![]\!] h \mu} & \text{SSA} & \xrightarrow{\mathcal{O}[\![]\!]} & \text{Imp} \\
\mathcal{I}[\![]\!](h,k)t \downarrow & & \downarrow \mathcal{E}[\![]\!]\sigma k & & \downarrow \mathcal{I}[\![]\!](h,k)t \\
v \in \mathcal{V} & = & v \in \mathcal{V} & = & v \in \mathcal{V}
\end{array}$$

new proof of Turing's Equivalence Theorem by compilation
(classical proof by simulation)

Future work: from prototyping back to implementation

- improve static profitability analysis of loop transformations
- can prototypes replace classical implementations?



Appendices

- recording denotational semantics of Imp
- denotational semantics of SSA
- compilation from Imp to SSA
- from SSA to MCR in PROLOG

Recording semantics of Imp

◂ Appendices

Syntax of Imp

Imp (a simple imperative language) is defined by:

$$
\begin{array}{ccl}
N & \in & Cst \\
I & \in & Ide \\
E & \in & Expr ::= N \mid I \mid E_1 \oplus E_2 \\
S & \in & Stmt ::= I = E \mid S_1 ; S_2 \mid \text{while}_\ell \; E \text{ do } S
\end{array}
$$

Backus-Naur Form (BNF) syntactic definitions

Recording semantics of Imp

point: $p = (h, k) \in P = N^* \times N^m$
states: $t \in T = Ide \rightarrow P \rightarrow \mathcal{V}$
semantics: $\mathcal{I}[\![]\!] \in Expr \rightarrow P \rightarrow T \rightarrow \mathcal{V}$

- evaluation point p is a pair (Dewey location, iteration vector)
- a state t yields for any identifier and evaluation point its numeric value in $\mathcal{V}$,
- the semantics $\mathcal{I}[\![]\!]$ expresses that an Imp expression, given a point and a state, denotes a value in $\mathcal{V}$

Recording semantics of Imp

$$\mathcal{I}[\![N]\!]pt \quad = \quad in\mathcal{V}(N)$$

semantics of a number is a value from $\mathcal{V}$.

Recording semantics of Imp

$$\mathcal{I}[\![N]\!]pt = in\mathcal{V}(N)$$
$$\mathcal{I}[\![I]\!]pt = R_{<p}(tI)$$

the imperative store semantics associates to a variable the last value assigned to it: $R_{<p}$ is used for reach the last definition at point p.

$$R_{<x}f = f(\max_{<x} Dom\ f)$$

Recording semantics of Imp

$$
\begin{aligned}
\mathcal{I}[\![N]\!]pt &= in\mathcal{V}(N) \\
\mathcal{I}[\![I]\!]pt &= R_{<p}(tI) \\
\mathcal{I}[\![E_1 \oplus E_2]\!]pt &= \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt
\end{aligned}
$$

decomposition of expressions

Recording semantics of Imp

$$
\begin{aligned}
\mathcal{I}[\![N]\!]pt &= in\mathcal{V}(N) \\
\mathcal{I}[\![I]\!]pt &= R_{<p}(tI) \\
\mathcal{I}[\![E_1 \oplus E_2]\!]pt &= \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt \\
\mathcal{I}[\![I = E]\!]pt &= t[\mathcal{I}[\![E]\!]pt/p/I]
\end{aligned}
$$

- for a statement and an iteration vector (at point p), record the computed value in the state t.
- collecting store semantics (only difference wrt classical imperative denotational semantics)

Recording semantics of Imp

$$
\begin{aligned}
\mathcal{I}[\![N]\!]pt &= in\mathcal{V}(N) \\
\mathcal{I}[\![I]\!]pt &= R_{<p}(tI) \\
\mathcal{I}[\![E_1 \oplus E_2]\!]pt &= \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt \\
\mathcal{I}[\![I = E]\!]pt &= t[\mathcal{I}[\![E]\!]pt/p/I] \\
\mathcal{I}[\![S_1; S_2]\!]p &= \mathcal{I}[\![S_2]\!](h.2, k) \circ \mathcal{I}[\![S_1]\!](h.1, k)
\end{aligned}
$$

composition of statements

Recording semantics of Imp

$$
\begin{aligned}
\mathcal{I}[\![N]\!]pt &= in\mathcal{V}(N)\\
\mathcal{I}[\![I]\!]pt &= R_{<p}(tI)\\
\mathcal{I}[\![E_1 \oplus E_2]\!]pt &= \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt\\
\mathcal{I}[\![I = E]\!]pt &= t[\mathcal{I}[\![E]\!]pt/p/I]\\
\mathcal{I}[\![S_1; S_2]\!]p &= \mathcal{I}[\![S_2]\!](h.2,k) \circ \mathcal{I}[\![S_1]\!](h.1,k)\\
\mathcal{I}[\![\text{while}_\ell\ E \text{ do } S]\!](h,k) &= \text{fix}(W)(h, k[0/\ell])
\end{aligned}
$$

$$
W = \lambda w.\lambda(h,k).\lambda t.\begin{cases} w(h, k_{\ell+})(\mathcal{I}[\![S]\!](h.1,k)t), & \text{if } \mathcal{I}[\![E]\!](h.1,k)t,\\ t, & \text{otherwise.}\end{cases}
$$

Classical least fixed point semantics for the loop.

Denotational semantics of SSA

Appendices

Syntax of SSA

SSA expressions are defined as follows:

$$\begin{array}{rcl} N & \in & Cst \\ I & \in & Ide \\ E & \in & SSA ::= N \mid I \mid E_1 \oplus E_2 \mid \mathrm{loop}_\ell\phi(E_1, E_2) \mid \mathrm{close}_\ell\phi(E_1, E_2) \end{array}$$

Backus-Naur Form (BNF) syntactic definitions

Denotational semantics of SSA

$$declarations : \sigma \in \Sigma = Ide_{\mathsf{SSA}} \rightarrow SSA$$

$$semantics : \mathcal{E}[\![]\!] \in SSA \rightarrow \Sigma \rightarrow N^m \rightarrow \mathcal{V}$$

- $\sigma \in \Sigma$ a map (identifier, expression)
- $\mathcal{E}[\![]\!]$ provides a value for an expression and an iteration vector

Denotational semantics of SSA

$$\mathcal{E}[\![N]\!]\sigma k \quad = \quad in\mathcal{V}(N)$$

semantics of a number is a value from $\mathcal{V}$.

Denotational semantics of SSA

$$\mathcal{E}[\![N]\!]\sigma k \quad = \quad in\mathcal{V}(N)$$
$$\mathcal{E}[\![I]\!]\sigma k \quad = \quad \mathcal{E}[\![\sigma I]\!]\sigma k$$

valuation of an identifier I is the valuation of its declaration in σ

Denotational semantics of SSA

$$
\begin{aligned}
\mathcal{E}[\![N]\!]\sigma k &= in\mathcal{V}(N) \\
\mathcal{E}[\![I]\!]\sigma k &= \mathcal{E}[\![\sigma I]\!]\sigma k \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma k &= \mathcal{E}[\![E_1]\!]\sigma k \oplus \mathcal{E}[\![E_2]\!]\sigma k
\end{aligned}
$$

decomposition of expressions

Denotational semantics of SSA

$$
\begin{array}{rcl}
\mathcal{E}[\![N]\!]\sigma k & = & in\mathcal{V}(N) \\
\mathcal{E}[\![I]\!]\sigma k & = & \mathcal{E}[\![\sigma I]\!]\sigma k \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma k & = & \mathcal{E}[\![E_1]\!]\sigma k \oplus \mathcal{E}[\![E_2]\!]\sigma k \\
\mathcal{E}[\![\mathrm{loop}_\ell \phi(E_1, E_2)]\!]\sigma k & = & \begin{cases} \mathcal{E}[\![E_1]\!]\sigma k, & \text{if } k_\ell = 0, \\ \mathcal{E}[\![E_2]\!]\sigma k_{\ell-}, & \text{otherwise.} \end{cases}
\end{array}
$$

primitive recursive declarations

Denotational semantics of SSA

$$
\begin{aligned}
\mathcal{E}[\![N]\!]\sigma k &= in\mathcal{V}(N) \\
\mathcal{E}[\![I]\!]\sigma k &= \mathcal{E}[\![\sigma I]\!]\sigma k \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma k &= \mathcal{E}[\![E_1]\!]\sigma k \oplus \mathcal{E}[\![E_2]\!]\sigma k \\
\mathcal{E}[\![\mathrm{loop}_\ell \phi(E_1, E_2)]\!]\sigma k &= \begin{cases} \mathcal{E}[\![E_1]\!]\sigma k, & \text{if } k_\ell = 0, \\ \mathcal{E}[\![E_2]\!]\sigma k_{\ell-}, & \text{otherwise.} \end{cases} \\
\mathcal{E}[\![\mathrm{close}_\ell \phi(E_1, E_2)]\!]\sigma k &= \mathcal{E}[\![E_2]\!]\sigma k[\min\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}/\ell]
\end{aligned}
$$

minimization operator $\Rightarrow$partial recursive declarations

SSA: a declarative language

$$
\begin{aligned}
\mathcal{E}[\![N]\!]\sigma k &= in\mathcal{V}(N) \\
\mathcal{E}[\![I]\!]\sigma k &= \mathcal{E}[\![\sigma I]\!]\sigma k \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma k &= \mathcal{E}[\![E_1]\!]\sigma k \oplus \mathcal{E}[\![E_2]\!]\sigma k \\
\mathcal{E}[\![\mathrm{loop}_\ell \phi(E_1, E_2)]\!]\sigma k &= \begin{cases} \mathcal{E}[\![E_1]\!]\sigma k, & \text{if } k_\ell = 0, \\ \mathcal{E}[\![E_2]\!]\sigma k_{\ell -}, & \text{otherwise.} \end{cases} \\
\mathcal{E}[\![\mathrm{close}_\ell \phi(E_1, E_2)]\!]\sigma k &= \mathcal{E}[\![E_2]\!]\sigma k[\min\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}/\ell]
\end{aligned}
$$

- no store update $\Rightarrow$not an imperative language
- no assignment in the "Static Single Assignment"
- SSA was misnamed

Compilation of Imp to SSA

◂ Appendices

A compiler for translating to SSA

$$
\begin{array}{ccc}
\text{Imp} & \xrightarrow{\mathcal{C}[\![]\!] h \mu} & \text{SSA} \\
\mathcal{I}[\![]\!](h,k)t \downarrow & & \downarrow \mathcal{E}[\![]\!]\sigma k \\
v \in \mathcal{V} & = & v \in \mathcal{V}
\end{array}
$$

$$\mathcal{C}[\![]\!] \in \text{Imp} \rightarrow N^* \rightarrow M \rightarrow SSA$$

$$\mu \in M = Ide \rightarrow N^* \rightarrow Ide_{\text{SSA}}$$

$\mathcal{C}[\![]\!]$ yields the SSA code corresponding to an imperative expression, given a Dewey identifier in N^* and a map μ between imperative and SSA identifiers

A compiler for translating to SSA

$$\mathcal{C}[\![N]\!]h\mu \quad = \quad N$$

numbers translated identically

A compiler for translating to SSA

$$\begin{aligned} \mathcal{C}[\![N]\!]h\mu &= N \\ \mathcal{C}[\![I]\!]h\mu &= R_{<h}(\mu I) \end{aligned}$$

last store to I translates to the reaching definition visible from h

A compiler for translating to SSA

$$
\begin{aligned}
\mathcal{C}[\![N]\!]h\mu &= N \\
\mathcal{C}[\![I]\!]h\mu &= R_{<h}(\mu I) \\
\mathcal{C}[\![E_1 \oplus E_2]\!]h\mu &= \mathcal{C}[\![E_1]\!]h\mu \oplus \mathcal{C}[\![E_2]\!]h\mu
\end{aligned}
$$

decomposition of expressions

A compiler for translating to SSA

$$
\begin{aligned}
\mathcal{C}[\![N]\!]h\mu &= N \\
\mathcal{C}[\![I]\!]h\mu &= R_{<h}(\mu I) \\
\mathcal{C}[\![E_1 \oplus E_2]\!]h\mu &= \mathcal{C}[\![E_1]\!]h\mu \oplus \mathcal{C}[\![E_2]\!]h\mu \\
\mathcal{C}[\![S_1; S_2]\!]h &= \mathcal{C}[\![S_2]\!]h.2 \circ \mathcal{C}[\![S_1]\!]h.1
\end{aligned}
$$

composition of statements

A compiler for translating to SSA

$$
\begin{array}{rcl}
\mathcal{C}[\![N]\!]h\mu & = & N \\
\mathcal{C}[\![I]\!]h\mu & = & R_{<h}(\mu I) \\
\mathcal{C}[\![E_1 \oplus E_2]\!]h\mu & = & \mathcal{C}[\![E_1]\!]h\mu \oplus \mathcal{C}[\![E_2]\!]h\mu \\
\mathcal{C}[\![S_1; S_2]\!]h & = & \mathcal{C}[\![S_2]\!]h.2 \circ \mathcal{C}[\![S_1]\!]h.1 \\
\mathcal{C}[\![I = E]\!]h(\mu, \sigma) & = & (\mu[I_h/h/I], \sigma[\mathcal{C}[\![E]\!]h\mu/I_h])
\end{array}
$$

define a new SSA identifier I_h for this store point h

A compiler for translating to SSA

$$
\begin{array}{rcl}
\mathcal{C}[\![\text{while}_\ell\ E \text{ do } S]\!]h(\mu,\sigma) & = & \theta_2 \text{ with} \\
\theta_0 & = & (\mu[I_{h.0}/h.0/I]_{I \in Dom\ \mu}, \\
& & \sigma[\text{loop}_\ell\phi(R_{<h}(\mu I), \bot)/I_{h.0}]_{I \in Dom\ \mu}), \\
\theta_1 & = & \mathcal{C}[\![S]\!]h.1\theta_0, \\
\theta_2 & = & (\mu_1[I_{h.2}/h.2/I]_{I \in Dom\ \mu_1}, \\
& & \sigma_1[\text{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))/I_{h.0}] \\
& & [\text{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})/I_{h.2}]_{I \in Dom\ \mu_1})
\end{array}
$$

- place loop and close phi nodes
- update map (μ) and SSA declarations (σ)

A compiler for translating to SSA

$$
\begin{aligned}
\mathcal{C}[\![\text{while}_\ell\ E \text{ do } S]\!]h(\mu,\sigma) &= \theta_2 \text{ with} \\
\theta_0 &= (\mu[I_{h.0}/h.0/I]_{I\in Dom\ \mu}, \\
&\quad \sigma[\text{loop}_\ell\phi(R_{<h}(\mu I),\bot)/I_{h.0}]_{I\in Dom\ \mu}), \\
\theta_1 &= \mathcal{C}[\![S]\!]h.1\theta_0, \\
\theta_2 &= (\mu_1[I_{h.2}/h.2/I]_{I\in Dom\ \mu_1}, \\
&\quad \sigma_1[\text{loop}_\ell\phi(R_{<h}(\mu I),R_{<h.2}(\mu_1 I))/I_{h.0}] \\
&\quad [\text{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})/I_{h.2}]_{I\in Dom\ \mu_1})
\end{aligned}
$$

- translate the loop body (θ_1)
- complete loop$_\ell\phi$ with the reaching definition from loop body

Consistency of translation

$$\begin{array}{ccc} \text{Imp} & \xrightarrow{\mathcal{C}[\![]\!] h \mu} & \text{SSA} \\ \mathcal{I}[\![]\!](h,k)t \downarrow & & \downarrow \mathcal{E}[\![]\!]\sigma k \\ v \in \mathcal{V} & = & v \in \mathcal{V} \end{array}$$

$$\mathcal{P}((\mu,\sigma),t,(h,k)) \Leftrightarrow \forall l \in Dom\ t, \mathcal{I}[\![l]\!]pt = \mathcal{E}[\![\mathcal{C}[\![l]\!]h\mu]\!]\sigma k$$

Theorem

Given $S \in Stmt$, *with* $(\mu,\sigma) = \mathcal{C}[\![S]\!]1\bot$, *and* $t = \mathcal{I}[\![S]\!](1,0^m)\bot$, $\mathcal{P}((\mu,\sigma),t,(2,0^m))$ *holds.*

consistency property holds after translating any Imp stmt

From SSA to MCR

◂ Appendices

From SSA to chains of recurrences

```
fromSSAtoMCR(ssa(X,lphi(_,_,X+Step)), ssa(X,unknown)) :-
  hasAself(X, Step) .
fromSSAtoMCR(ssa(X, lphi(LoopId, Init, X + Step)),
             ssa(X, mcr(LoopId, Init, Step))).

hasAself(X, X).
hasAself(X, Name, Step) :- ssa(Name, Expr), hasAself(X, Expr).
hasAself(X, A + B) :- hasAself(X, A); hasAself(X, B).
```

- some information is lost
- there is no computation
- use the unification engine of PROLOG

From SSA to Lambda functions

```
fromSSAtoLambda(ssa(X,lphi(_,_,X+Step)),ssa(X, unknown)) :-
  hasAself(X,Step).
fromSSAtoLambda(ssa(X,lphi(LoopId,Init,X+Step)),
            ssa(X,Init+Result)) :-
  sumNFirst(LoopId,LoopId,Step,Result).
```

```
sumNFirst(L,N,C*lambda(L,binom(L,K)),C*lambda(N,binom(N,K1))) :-
  integer(C), fold(K+1,K1).
```

$$\sum_{L=0}^{N-1} C \cdot \binom{L}{K} = C \cdot \binom{N}{K+1}$$

Scalar evolutions: abstractions extracted from SSA

- SSA $\rightarrow$MCR $\rightarrow$Lambda
- SSA $\rightarrow$Lambda

we have seen the semantics of a subset of the SSA
(by mappings to polynomial lambda expressions)