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
The SSA Representation Framework: Semantics, Analyses and GCC Implementation

Structure of Modern Compilers

Front-end

C  C++  Java  F95  Ada

Middle-end

Generic Imperative Language

Static Single Assignment (SSA)

Back-end

Machine Description

x86  ppc  ia64  arm  mips

Normalization

Normalization

Specialization

SSA used for reducing the complexity of static scalar analyses
SSA Representation

- 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:

\[
a = 1 \\
do \text{ forever} \\
a = a + 3
\]
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:

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
\[ c = \text{phi}(a, f) \]
\[ d = \text{phi}(b, g) \]
\[ \text{if}(d > 123) \text{ goto end} \]
\[ e = d + 7 \]
\[ f = e + c \]
\[ g = d + 5 \]
endloop
end:

Found the starting loop-phi. The SCC is:

\[ c \rightarrow f \rightarrow c \]
Example: finding the evolution of scalar “c”

\[ a = 3 \]
\[ b = 1 \]
\[ \text{loop} \]
\[ c = \text{phi} \ (a, \ f) \]
\[ d = \text{phi} \ (b, \ g) \]
\[ \text{if} \ (d > 123) \ \text{goto end} \]
\[ e = d + 7 \]
\[ f = e + c \]
\[ g = d + 5 \]
\[ \text{endloop} \]
\[ \text{end:} \]

Reconstruct the update expression: \( c + e \)

\[ c = \text{phi} \ (a, \ c + e) \rightarrow \{a, +, e\} \]
Example: finding the evolution of scalar “c”

\[
\begin{align*}
a &= 3 \\
b &= 1 \\
\text{loop} \\
c &= \text{phi} (a, f) \\
d &= \text{phi} (b, g) \\
\text{if} (d > 123) \text{ goto end} \\
e &= d + 7 \\
f &= e + c \\
g &= d + 5 \\
\text{endloop} \\
\text{end:}
\end{align*}
\]

\[
c \rightarrow \{a, +, e\} \xrightarrow{\text{Instantiate}} \text{Optional} \ldots
\]
Example: finding the evolution of scalar “c”

\[ a = 3 \]
\[ b = 1 \]
\[ \text{loop} \]
\[ c = \phi (a, f) \]
\[ d = \phi (b, g) \]
\[ \text{if } (d > 123) \text{ goto end} \]
\[ e = d + 7 \]
\[ f = e + c \]
\[ g = d + 5 \]
\[ \text{endloop} \]
\[ \text{end:} \]

\[ c \rightarrow \{a, +, e\} \xrightarrow{\text{Instantiate}} \{3, +, e\} \]
Example: finding the evolution of scalar “c”

\[ a = 3 \]
\[ b = 1 \]

\[ \text{loop} \]
\[ c = \text{phi} (a, f) \]
\[ d = \text{phi} (b, g) \]
\[ \text{if} (d > 123) \text{ goto end} \]
\[ e = d + 7 \]
\[ f = e + c \]
\[ g = d + 5 \]
\[ \text{endloop} \]
\[ \text{end:} \]

\[ c \rightarrow \{ a, +, e \} \xrightarrow{\text{Instantiate}} \{ 3, +, e \} \]
\[ e \rightarrow d + 7 \]
Example: finding the evolution of scalar “c”

\[
a = 3 \\
b = 1 \\
\text{loop} \\
c = \text{phi}(a, f) \\
d = \text{phi}(b, g) \\
\text{if (d > 123) goto end} \\
e = d + 7 \\
f = e + c \\
g = d + 5 \\
\text{endloop} \\
\text{end:}
\]

\[
c \rightarrow \{a, +, e\} \xrightarrow{\text{Instantiate}} \{3, +, e\} \\
e \rightarrow d + 7 \\
d \rightarrow \{1, +, 5\}
\]
Example: finding the evolution of scalar “c”

\[
a = 3 \\
b = 1 \\
loop \\
c = \text{phi} (a, f) \\
d = \text{phi} (b, g) \\
\text{if} (d > 123) \text{ goto end} \\
e = d + 7 \\
f = e + c \\
g = d + 5 \\
endloop
eend:
\]

\[
c \rightarrow \{a, +, e\} \xrightarrow{\text{Instantiate}} \{3, +, e\} \\
e \rightarrow \{8, +, 5\} \\
d \rightarrow \{1, +, 5\}
\]
Example: finding the evolution of scalar “c”

\[
\begin{align*}
a &= 3 \\
b &= 1 \\
\text{loop} \\
c &= \text{phi}(a, f) \\
d &= \text{phi}(b, g) \\
\text{if } (d > 123) \quad \text{goto end} \\
e &= d + 7 \\
f &= e + c \\
g &= d + 5 \\
\text{endloop} \\
\end{align*}
\]

\[
c \rightarrow \{a, +, e\} \xrightarrow{\text{Instantiate}} \{3, +, 8, +, 5\}(x) = 3^0 x + 8^1 x + 5^2 x \\
e \rightarrow \{8, +, 5\}
\]
Applications

\[
\begin{align*}
&b = 1 \\
&\text{loop} \\
&\quad d = \text{phi} \ (b, \ g) \\
&\quad \text{if} \ (d > 123) \ \text{goto end} \\
&\quad g = d + 5 \\
&\text{endloop} \\
&\text{end:} \\
&h = \text{phi} \ (d)
\end{align*}
\]

\[
\begin{align*}
&b = 1 \\
&\text{loop} \\
&\quad d = \text{phi} \ (b, \ g) \\
&\quad \text{if} \ (d > 123) \ \text{goto end} \\
&\quad g = d + 5 \\
&\text{endloop} \\
&\text{end:} \\
&h = 126
\end{align*}
\]

- 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:

\[ N \in Cst \]
\[ I \in Ide \]
\[ E \in SSA ::= N \mid I \mid E_1 \oplus E_2 \mid \text{loop}_\ell\phi(E_1, E_2) \mid \text{close}_\ell\phi(E_1, E_2) \]
Denotational semantics of SSA

- associate an expression to each SSA identifier, \( \sigma: I \rightarrow E \)
- iteration vectors, \( k \in N^m \)

\[
E[\text{loop}_{\ell} \phi(E_1, E_2)] \sigma k = \begin{cases} 
E[E_1] \sigma k, & \text{if } k_\ell = 0, \\
E[E_2] \sigma k_{\ell^-}, & \text{otherwise.}
\end{cases}
\]

\[
E[\text{close}_{\ell} \phi(E_1, E_2)] \sigma k = E[E_2] \sigma k[\min\{x \mid -E[E_1] \sigma k[x/\ell]\}]/\ell
\]

- \( \text{loop}\phi \) provides values for some \( k \) (primitive recursive)
- \( \text{close}\phi \) 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:

\[ N \in Cst \]
\[ I \in Ide \]
\[ E \in Expr ::= N \mid I \mid E_1 \oplus E_2 \]
\[ S \in Stmt ::= I = E \mid S_1 ; S_2 \mid \text{while}\_\ell \ E \ \text{do} \ S \]

- loops uniquely identified by \( \ell \)
- 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

\[ K = 7; \]
\[ J = 0; \]
\[ \text{while}_{1}(J < 10) \text{ do} \]
\[ J = J + K; \]

\[ K = 7; \]
\[ J = 0; \]
\[ \text{while}_{1}(J < 10) \text{ do} \]
\[ J = J + K; \]
\[ \text{while}_{2}(J < 100) \text{ do} \]
\[ J = J + K; \]

\[ (\ell_1) : (0) \prec (1) \prec (2) \prec (2) \prec \ldots \prec (2_{13}) \]
Consistency of translation

\[
\text{Imp} \xrightarrow{C \llbracket h\mu} \text{SSA}
\]

\[
\llbracket (h,k) t \rrbracket \downarrow \quad \llbracket \sigma_k \rrbracket
\]

\[\nu \in \mathcal{V} \quad \nu \in \mathcal{V}\]

consistency property holds after translating any Imp stmt
Part 3: Abstract SSA in PROLOG
SSA in PROLOG

\[ K = 7; \]
\[ J = 0; \]
\[ \text{while} \ (J < 10) \ \text{do} \]
\[ J = J + K; \]

\[ \text{PROLOG} \]
\[ \text{ssa}(K, 7). \]
\[ \text{ssa}(J, 0). \]
\[ \text{ssa}(A, \text{lphi}(11, J, B)). \]
\[ \text{ssa}(B, A + K). \]
\[ \text{ssa}(C, \text{cphi}(11, A < 10, A)). \]

SSA declarations represented by PROLOG facts
From SSA to chains of recurrences

\[
\text{fromSSAtoMCR}(\text{ssa}(X, \text{lphi}(-, -, X+\text{Step})), \text{ssa}(X, \text{unknown})) :- \\
\text{hasAself}(X, \text{Step}).
\]

\[
\text{fromSSAtoMCR}(\text{ssa}(X, \text{lphi}(\text{LoopId}, \text{Init}, X + \text{Step})), \\
\text{ssa}(X, \text{mcr}(\text{LoopId}, \text{Init}, \text{Step}))).
\]

\[
\text{hasAself}(X, X).
\]

\[
\text{hasAself}(X, \text{Name}, \text{Step}) :- \text{ssa}(\text{Name}, \text{Expr}), \text{hasAself}(X, \text{Expr}).
\]

\[
\text{hasAself}(X, A + B) :- \text{hasAself}(X, A); \text{hasAself}(X, B).
\]

- 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


- High-Level Loop Optimizations for GCC. Daniel Berlin, David Edelsohn (IBM T.J. Watson Research Center), Sebastian Pop. GCC Summit 2004, Ottawa, Canada.

Future work: out of SSA

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

new proof of Turing’s Equivalence Theorem by \textit{compilation} (classical proof by \textit{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
Syntax of Imp

Imp (a simple imperative language) is defined by:

\[ N \in Cst \]
\[ I \in Ide \]
\[ E \in Expr ::= N \mid I \mid E_1 \oplus E_2 \]
\[ S \in Stmt ::= I = E \mid S_1; S_2 \mid \text{while}_\ell E \text{ do } S \]

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: \( I[] \in Expr \rightarrow P \rightarrow T \rightarrow \mathcal{V} \)

- 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 \( I[] \) expresses that an Imp expression, given a point and a state, denotes a value in \( \mathcal{V} \)
Recording semantics of Imp

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

semantics of a number is a value from \( \mathcal{V} \).
Recording semantics of Imp

\[ \mathcal{I}[N]pt = \text{in}V(N) \]
\[ \mathcal{I}[l]pt = R_{<p}(tl) \]

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 \( p \).

\[ R_{<x}f = f(\max_{<x}\text{Dom } f) \]
Recording semantics of Imp

\[ \mathcal{I}[N]_{pt} = \text{in}\mathcal{V}(N) \]

\[ \mathcal{I}[I]_{pt} = R_{<p}(t) \]

\[ \mathcal{I}[E_1 \oplus E_2]_{pt} = \mathcal{I}[E_1]_{pt} \oplus \mathcal{I}[E_2]_{pt} \]

decomposition of expressions
Recording semantics of Imp

\[
\begin{align*}
I[N]_{pt} &= inV(N) \\
I[I]_{pt} &= R_{<p}(tl) \\
I[E_1 \oplus E_2]_{pt} &= I[E_1]_{pt} \oplus I[E_2]_{pt} \\
I[I = E]_{pt} &= t[I[E]_{pt}/p/l]
\end{align*}
\]

- 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

\[ \mathcal{I}[N]pt = \text{inY}(N) \]
\[ \mathcal{I}[l]pt = R_{<p}(tl) \]
\[ \mathcal{I}[E_1 \oplus E_2]pt = \mathcal{I}[E_1]pt \oplus \mathcal{I}[E_2]pt \]
\[ \mathcal{I}[l = E]pt = t[\mathcal{I}[E]pt/p/l] \]
\[ \mathcal{I}[S_1; S_2]p = \mathcal{I}[S_2](h.2, k) \circ \mathcal{I}[S_1](h.1, k) \]

composition of statements
Recording semantics of Imp

\[ \begin{align*}
\mathcal{I}[N] & = \text{inV}(N) \\
\mathcal{I}[I] & = R_{<p}(tI) \\
\mathcal{I}[E_1 \oplus E_2] & = \mathcal{I}[E_1] \oplus \mathcal{I}[E_2] \\
\mathcal{I}[l = E] & = t[\mathcal{I}[E] / p / l] \\
\mathcal{I}[S_1 ; S_2] & = \mathcal{I}[S_2](h.2, k) \circ \mathcal{I}[S_1](h.1, k) \\
\mathcal{I}[\text{while}_l E \text{ do } S](h, k) & = \text{fix}(W)(h, k[0/l])
\end{align*} \]

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

Classical least fixed point semantics for the loop.
Denotational semantics of SSA
Syntax of SSA

SSA expressions are defined as follows:

\[ N \in \text{Cst} \]
\[ I \in \text{Ide} \]
\[ E \in \text{SSA} ::= N \mid I \mid E_1 \oplus E_2 \mid \text{loop}_\ell \phi(E_1, E_2) \mid \text{close}_\ell \phi(E_1, E_2) \]

Backus-Naur Form (BNF) syntactic definitions
Denotational semantics of SSA

\[
\text{declarations} : \sigma \in \Sigma = \text{Id}_{\text{SSA}} \rightarrow \text{SSA}
\]

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

- $\sigma \in \Sigma$ a map (identifier, expression)
- $E[]$ provides a value for an expression and an iteration vector
Denotational semantics of SSA

\[ \mathcal{E}[N]_{\sigma k} = \text{in}\mathcal{V}(N) \]

semantics of a number is a value from \( \mathcal{V} \).
Denotational semantics of SSA

\[ \mathcal{E}[N]_{\sigma k} = \text{in}(N) \]
\[ \mathcal{E}[l]_{\sigma k} = \mathcal{E}[\sigma l]_{\sigma k} \]

valuation of an identifier \( l \) is the valuation of its declaration in \( \sigma \)
Denotational semantics of SSA

\[
\begin{align*}
\mathcal{E}[N]_{\sigma k} &= \text{in}\mathcal{V}(N) \\
\mathcal{E}[l]_{\sigma k} &= \mathcal{E}[\sigma l]_{\sigma k} \\
\mathcal{E}[E_1 \oplus E_2]_{\sigma k} &= \mathcal{E}[E_1]_{\sigma k} \oplus \mathcal{E}[E_2]_{\sigma k}
\end{align*}
\]

decomposition of expressions
Denotational semantics of SSA

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

primitive recursive declarations
Denotational semantics of SSA

\[ E[N]σ_k = \text{in}\mathcal{V}(N) \]
\[ E[I]σ_k = E[σI]σ_k \]
\[ E[E_1 ⊕ E_2]σ_k = E[E_1]σ_k ⊕ E[E_2]σ_k \]
\[ E[\text{loop}_ℓφ(E_1, E_2)]σ_k = \begin{cases} E[E_1]σ_k, & \text{if } k_ℓ = 0, \\ E[E_2]σ_{k_ℓ}, & \text{otherwise.} \end{cases} \]
\[ E[\text{close}_ℓφ(E_1, E_2)]σ_k = E[E_2]σ_k[\min\{x \mid \neg E[E_1]σ_k[x/ℓ]\}/ℓ] \]

minimization operator ⇒ partial recursive declarations
SSA: a declarative language

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

- no store update ⇒ not an imperative language
- no assignment in the “Static Single Assignment”
- SSA was misnamed
Compilation of Imp to SSA
A compiler for translating to SSA

\[ \text{Imp} \xrightarrow{C[]} h\mu \rightarrow \text{SSA} \]

\[ \mathcal{I}(h,k)t \downarrow \quad \mathcal{E}\sigma k \]

\[ v \in \mathcal{V} \]

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

\[ \mu \in M = \text{Ide} \rightarrow N^* \rightarrow \text{Ide}_{\text{SSA}} \]

\[ C[] \text{ yields the SSA code corresponding to an imperative expression, given a Dewey identifier in } N^* \text{ and a map } \mu \text{ between imperative and SSA identifiers} \]
A compiler for translating to SSA

\[ C[[N]] h_\mu = N \]

numbers translated identically
A compiler for translating to SSA

\[
C[N] h\mu = N \\
C[I] h\mu = R_{<h}(\mu I)
\]

last store to \( I \) translates to the reaching definition visible from \( h \)
A compiler for translating to SSA

\[
\begin{align*}
C[N]h_\mu &= N \\
C[I]h_\mu &= R_{<h}(\mu I) \\
C[E_1 \oplus E_2]h_\mu &= C[E_1]h_\mu \oplus C[E_2]h_\mu
\end{align*}
\]

decomposition of expressions
A compiler for translating to SSA

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

composition of statements
A compiler for translating to SSA

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

define a new SSA identifier \( I_h \) for this store point \( h \)
A compiler for translating to SSA

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

- place loop and close phi nodes
- update map (\(\mu\)) and SSA declarations (\(\sigma\))
A compiler for translating to SSA

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

- translate the loop body (\theta_1)
- complete loop \phi with the reaching definition from loop body
Consistency of translation

\[ \text{Imp} \xrightarrow{\mathcal{C}\llbracket h\mu \rrbracket} \text{SSA} \]

\[ \mathcal{I}\llbracket (h,k) t \rrbracket \downarrow \mathcal{E}\llbracket \sigma k \rrbracket \]

\[ \forall v \in \mathcal{V} \]

\[ \mathcal{P}((\mu, \sigma), t, (h, k)) \iff \forall I \in \text{Dom } t, \mathcal{I}\llbracket I \rrbracket pt = \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h\mu \rrbracket \sigma k \]

**Theorem**

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

Consistency property holds after translating any Imp stmt
From SSA to MCR
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

\[
\text{fromSSAtolambda}(\text{ssa}(X, \text{lphi}(-, -, X+\text{Step})), \text{ssa}(X, \text{unknown})) :\text{-}
\text{hasAsel}(X, \text{Step}).
\]

\[
\text{fromSSAtolambda}(\text{ssa}(X, \text{lphi}(\text{LoopId}, \text{Init}, X+\text{Step})),
\text{ssa}(X, \text{Init}+\text{Result})) :\text{-}
\text{sumNFirst}(\text{LoopId}, \text{LoopId}, \text{Step}, \text{Result}).
\]

\[
\text{sumNFirst}(L, N, C*\text{lambda}(L, \text{binom}(L, K)), C*\text{lambda}(N, \text{binom}(N, K1))) :\text{-}
\text{integer}(C), \text{fold}(K+1, K1).
\]

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