In and Out of SSA: A Denotational Specification[1]

Sebastian Pop

Open Source Compiler Engineering - Advanced Micro Devices Inc.
Austin, Texas

SSA Seminar - Autrans, France - April 27, 2009

[1] Joint work with Pierre Jouvelot and Georges-André Silber (CRI, MINES ParisTech)

Outline

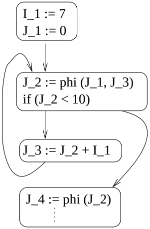
- SSA: a declarative language and its denotational semantics
- compilers from Imp to SSA and back
- SSA slices Imp: parallelization and compression of Imp
- from RAM computing model to Kleene's partial recursive functions: a new reduction by compilation

SSA: Static Single Assignment Representation

- internal compiler representation
- annotations on top of an imperative representation

```
I := 7;
J := 0;
while J < 10 do
   J := J + I;
end
```

Classical SSA ⟶



- (control flow) graph based representation
- explicit use-def links
- phi nodes at control flow junctions

Motivation for a Denotational Specification of the SSA

Provide a firm foundation for the SSA representation

- define a language for the SSA: syntax and semantics
- prove consistency properties of conversion to and out of SSA
- opening a new venue to formal program analysis techniques, such as abstract interpretation, to operate on SSA

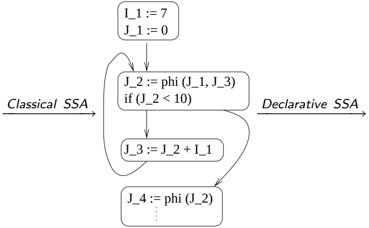
A Declarative Language Definition for SSA

- BNF syntax

$$
\begin{array}{lll}
I_h & \in & Ide_{\mathrm{SSA}} \\
E & \in & SSA ::= N \mid I_h \mid E_0 \oplus E_1 \mid \mathrm{loop}_h(E_0, E_1) \mid \mathrm{close}_h(E_0, E_1)
\end{array}
$$

- example

```
I := 7;
J := 0;
while J < 10 do
  J := J + I;
end
```



$$
\left\{
\begin{array}{lll}
I_1 & \longrightarrow & 7 \\
J_1 & \longrightarrow & 0 \\
J_2 & \longrightarrow & \mathrm{loop}_1(J_1, J_3) \\
J_3 & \longrightarrow & J_2 + I_1 \\
J_4 & \longrightarrow & \mathrm{close}_1(J_2 < 10, J_2)
\end{array}
\right.
$$

- drop: assignments, sequence, gotos, basic blocks, edges, loops everything belonging to an imperative programming language
- keep: an unordered map of (identifier, expression) defining the same computation

A Declarative Language Definition for SSA

- denotational semantics

$$
\begin{aligned}
h &\in N^* \\
\rho &\in Ide_{\mathsf{SSA}} \to K \to \mathcal{V} \\
k &\in K = N^* \to N \\
\mathcal{E}[\![I]\!]\rho k &= \rho I k \\
\mathcal{E}[\![\mathrm{loop}_h(E_0, E_1)]\!]\rho k &= \begin{cases} \mathcal{E}[\![E_0]\!]\rho k, \text{if } kh = 0, \\ \mathcal{E}[\![E_1]\!]\rho k_{h-}, \text{otherwise.} \end{cases} \\
\mathcal{E}[\![\mathrm{close}_h(E_0, E_1)]\!]\rho k &= \mathcal{E}[\![E_1]\!]\rho k[\min\{x \mid \neg\mathcal{E}[\![E_0]\!]\rho k[x/h]\}/h]
\end{aligned}
$$

- example

$$
\sigma : \begin{cases} I_1 & \to & 7 \\ J_1 & \to & 0 \\ J_2 & \to & \mathrm{loop}_1(J_1, J_3) \\ J_3 & \to & J_2 + I_1 \\ J_4 & \to & \mathrm{close}_1(J_2 < 10, J_2) \end{cases}
\xrightarrow{\mathcal{E}[\![]\!](\mathcal{R}\sigma)k}
\rho : \begin{cases} I_1 & \to & \lambda k.7 \\ J_1 & \to & \lambda k.0 \\ J_2 & \to & \lambda k. \begin{cases} J_1(k) \text{ for } k1 = 0 \\ J_3(k_{1-}) \text{ for } k1 > 0 \end{cases} \\ J_3 & \to & \lambda k. J_2(k) + I_1(k) \\ J_4 & \to & \lambda k.14 \end{cases}
$$

Compilers from Imp to SSA and Back

- definition of Imp
- conversion to SSA
- out-of-SSA

Imp: the Simple Imperative Language (from Textbooks)

- BNF syntax

$$
\begin{array}{lcl}
N & \in & Cst \\
I & \in & Ide \\
E & \in & Expr ::= N \mid I \mid E_0 \oplus E_1 \\
S & \in & Stmt ::= I := E \mid S_0 ; S_1 \mid \text{while } E \text{ do } S \text{ end}
\end{array}
$$

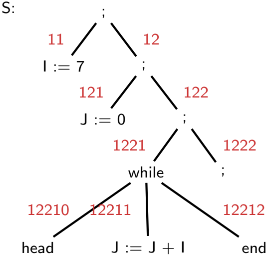
- Dewey numbering

$$
\begin{array}{rcl}
n+ & = & n+1 \\
(h.n)+ & = & h.(n+1) \quad (1 \leq n < m) \\
(h.m)+ & = & h+
\end{array}
$$

- example: AST (Abstract Syntax Tree) and Dewey numbering

S:
```
I := 7;
J := 0;
while J < 10 do
  J := J + I;
end
```

AST Dewey numbering →



Imp: the Simple Imperative Language (from Textbooks)

- "run-time evaluation point" combination of syntactic h and dynamic k information: $p = (h, k) \in P = N^* \times K$
- reaching definitions: $R_{<x} f = f(\max_{<x} Dom\ f)$
- $\mathcal{I}[\![]\!]$ interpreter state $t \in T = Ide \rightarrow P \rightarrow \mathcal{V}$
- denotational semantics of Imp:

$$
\begin{array}{rcl}
\mathcal{I}[\![I]\!]pt & = & R_{<p}(tI) \\
\mathcal{I}[\![I := E]\!]h(k,t) & = & (k, t[\mathcal{I}[\![E]\!](h,k)t/(h,k)/I]) \\
\mathcal{I}[\![\text{while } E \text{ do } S \text{ end}]\!]h(k,t) & = & \text{fix}(W_h)(k[0/h], t) \\
W_h & = & \lambda w.\lambda u. \begin{cases} w(k'_{h+}, t'), & \text{if } \mathcal{I}[\![E]\!](h.1, k)t, \\ u, & \text{otherwise.} \end{cases} \\
 & & \text{where } (k', t') = \mathcal{I}[\![S]\!]h.1 \text{ and } (k,t) = u
\end{array}
$$

$\mathcal{C}[\![]\!]$: Conversion from Imp to SSA

- conversion state $\mu \in M = Ide \rightarrow N^* \rightarrow Ide_{\text{SSA}}$ maps Imp identifiers to SSA identifiers at given Dewey points h

$$\mathcal{C}[\![I]\!]h\mu \quad = \quad R_{<h}(\mu I)$$

- For every Imp variable I and for every Dewey point h in an Imp program, $\mathcal{C}[\![]\!]$ provides an SSA identifier I_h and an expression $\mathcal{C}[\![E]\!]h\mu$ that computes the value of the variable at that location:

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

$\mathcal{C}[\![]\!]$: Example of Conversion from Imp to SSA

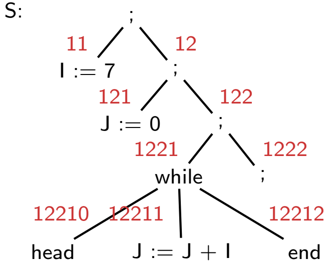
- conversion example

S:
```
I := 7;
J := 0;
while J < 10 do
  J := J + I;
end
```

$\xrightarrow{\mathcal{C}[\![]\!]1\bot}$

σ:
```
I_11, 7
I_12210, loop_1221(I_11, I_12210)
I_12212, close_1221(J_12210 < 10, I_12210)
J_121, 0
J_12210, loop_1221(J_121, J_122111)
J_122111, J_12210 + I_12210
J_12212, close_1221(J_12210 < 10, J_12210)
```

- Dewey numbering from 1 of the AST



SSA slices Imp

- via the SSA conversion process, the standard semantics for expression gets staged, informally getting "curryied" from $Expr \rightarrow (N^* \times K) \rightarrow T \rightarrow \mathcal{V}$ to $Expr \rightarrow N^* \rightarrow K \rightarrow T \rightarrow \mathcal{V}$

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

- a profound perspective change: uncoupling of syntactic sequencing h from run time iteration space sequencing k

$\mathcal{O}[\![]\!]$: Out-of-SSA

For an identifier I and an SSA expression E, returns the Imp code required to compute "$I := E$"

$$\mathcal{O}[\![I, I']\!]\sigma a \kappa = \text{up}[\![I, I := I']\!]a\kappa_0, \text{with } \kappa_0 = \begin{cases} \mathcal{O}[\![I', \sigma I']\!]\sigma a(\text{up}_{\text{env}}[\![I']\!]a\kappa), & \text{if } I' \notin \text{dom}_{\text{env}}(\kappa) \\ \kappa, \text{otherwise}, \end{cases}$$

$$\mathcal{O}[\![I, \text{loop}_h(E_0, E_1)]\!]\sigma a \kappa = \text{up}[\![I, I := I_1]\!](h, k)\kappa_1, \text{with } \kappa_1 = (\mathcal{O}[\![I_1, E_1]\!]\sigma(h, \text{body}) \circ \mathcal{O}[\![I, E_0]\!]\sigma(h, \text{head}))\kappa,$$

$$\mathcal{O}[\![I, \text{close}_h(E_0, E_1)]\!]\sigma a \kappa = \text{up}[\![I, \kappa_1(h, \text{head}); \text{while } I_0 \text{ do } \kappa_1(h, \text{body}); \kappa_1(h, k) \text{ end}; I := I_1]\!]a\kappa_1$$
$$\text{with } \kappa_1 = (\mathcal{O}[\![I_1, \text{loop}_h(E_1, E_1)]\!]\sigma a \circ \mathcal{O}[\![I_0, \text{loop}_h(E_0, E_0)]\!]\sigma a)\kappa$$

- extracts from σ a slice computing only the required identifier
- parallelization: $\mathcal{O}[\![]\!]$ may independently generate as many Imp programs as identifiers in σ
- future work: eliminate redundant computations: compression (needed for the efficient operation of sequential processors)

$\mathcal{O}[\![]\!]$: Out-of-SSA Example

Output of our GNU Common Lisp prototype, when requesting the value of J_4 to be stored in the fresh variable `RESULT`:

$$\sigma:\ \begin{array}{l} I_1 \rightarrow 7 \\ J_1 \rightarrow 0 \\ J_2 \rightarrow \text{loop}_1(J_1,\ J_3) \\ J_3 \rightarrow J_2 + I_1 \\ J_4 \rightarrow \text{close}_1(J_2 < 10,\ J_2) \end{array} \xrightarrow{(\mathcal{O}[\![\texttt{RESULT},\ \texttt{J}_4]\!]\sigma(0,\text{head})\bot)(0,\text{head})} \text{S:}$$

```
J1 := 0;
J2 := J1;
I0_33062 := J2;
I1_33062 := 10;
I0_33060 := I0_33062 < I1_33062;
I1_33060 := J2;
while I0_33060 do
  I0_33068 := J2;
  I1 := 7;
  I1_33068 := I1;
  J3 := I0_33068 + I1_33068;
  I1_33064 := J3;
  I0_33073 := J2;
  I1_33073 := 10;
  I1_33061 := I0_33073 < I1_33073;
  I1_33076 := J2;
  J2 := I1_33064;
  I0_33060 := I1_33061;
  I1_33060 := I1_33076;
end
J4 := I1_33060;
RESULT := J4;
```

From RAM to Partial Recursive Functions (by Compilation)

- Imp is equivalent to RAM
- SSA is equivalent to Kleene's Partial Recursive Functions

$$
\begin{array}{rcl}
\mathcal{K}[\![I]\!]x & = & I(x) \\
\mathcal{K}[\![I, \mathrm{loop}_h(E_0, E_1)]\!]x & = & \{I(x_{1,h-1}, 0, x_{h+1,m}) = \mathcal{K}[\![E_0]\!](x_{1,h-1}, 0, x_{h+1,m}), \\
 & & \ I(x_{1,h-1}, z+1, x_{h+1,m}) = \mathcal{K}[\![E_1]\!](x_{1,h-1}, z, x_{h+1,m})\} \\
\mathcal{K}[\![I, \mathrm{close}_h(E_0, E_1)]\!]x & = & \{\mathrm{min}_I(x_{1,h-1}, x_{h+1,m}) = (\mu y.\mathcal{K}[\![E_0]\!](x_{1,h-1}, y, x_{h+1,m}) = 0), \\
 & & \ I(x) = \mathcal{K}[\![E_1]\!](x_{1,h-1}, \mathrm{min}_I(x_{1,h-1}, x_{h+1,m}), x_{h+1,m})\}
\end{array}
$$

- proofs of translation consistency from Imp to SSA and back (see our paper)
- new proof of Turing's Equivalence Theorem between RAM and Partial Recursive Functions, previously typically proven using simulation [JonesND1997].

Conclusion

- new language definition for the SSA
- new insights on the essence of the SSA
- new venue for abstract interpretation based on SSA
- new proof of Turing's Equivalence Theorem by compilation

Research report: http://cri.ensmp.fr/classement/doc/E-285.pdf