SSA and Psi-SSA Representations

2009/04/15

François de Ferrière, Christophe Guillon

Francois.de-ferriere@st.com, christophe.guillon@st.com

The SSA form

- SSA form presentation outline:
  - SSA form definition
  - SSA form properties
  - SSA construction
  - SSA destruction
  - Optimisations over SSA form programs

SSA Form: Static Single Assignment

- Every variable as exactly one static definition

$x=2$
$y=x+1$
$x=3$
$z=x+2$

not SSA

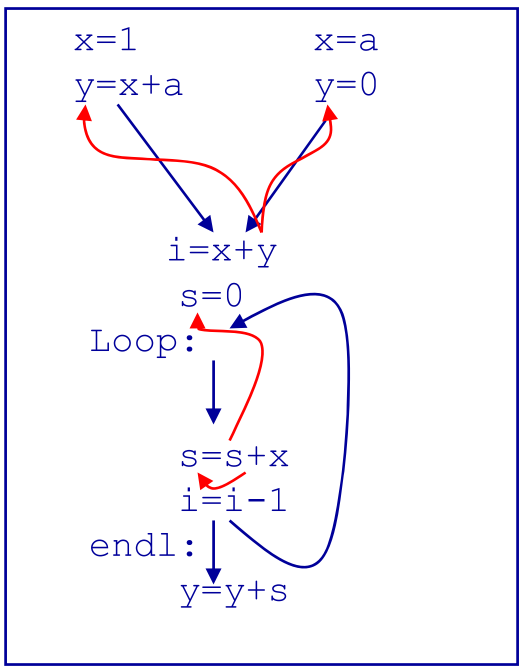
$x_1=2$
$y=x_1+1$
$x_2=3$
$z=x_2+2$

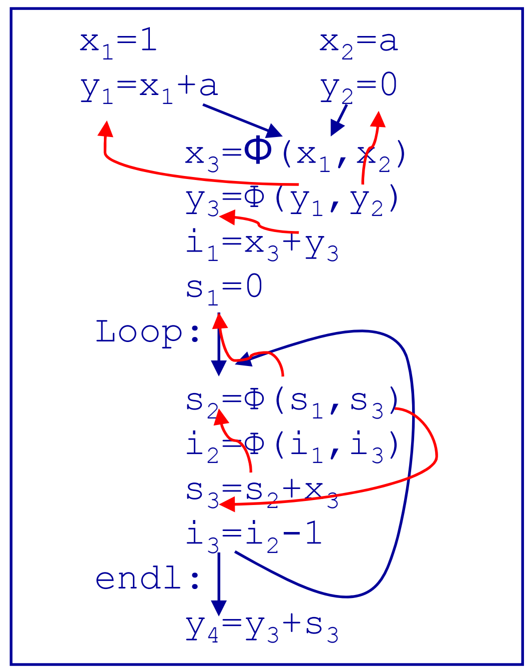
SSA

- It is a property of the program, not a new IR
- Motivation
  - Identify variable name and defining operation
  - Single reaching definition made explicit

SSA Property: Single Definition

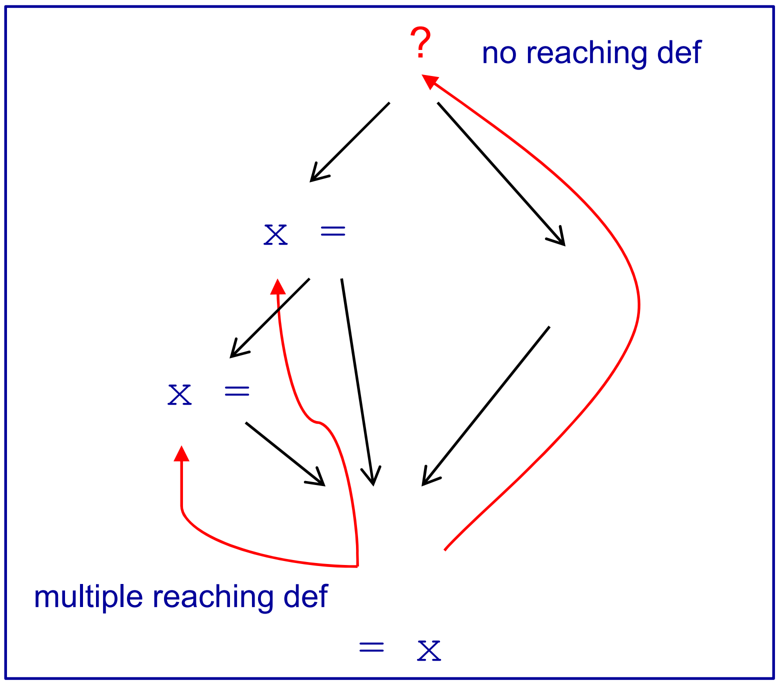
- Each assignment to a variable is given a unique name (at most one definition):
  - Simple renaming for straight-line code
  - Φ–nodes are introduced on control-flow join points



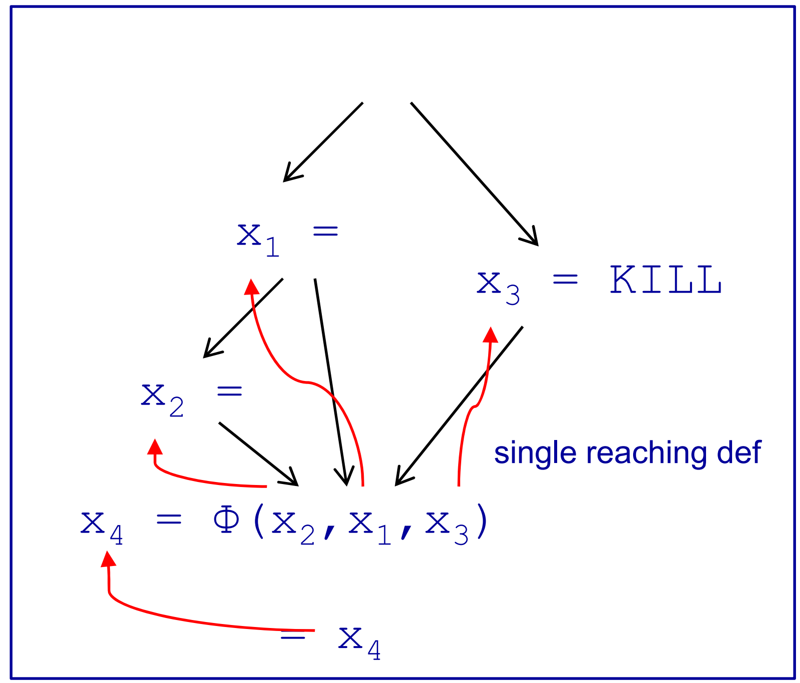


SSA Property: Reaching definition

- Each use has a reaching definition (at least one definition):
  - KILL nodes inserted to enforce definition



not SSA



SSA

SSA Properties

- More compact representation than def-use chains
- Information on a variable is true everywhere (independent from the Control-Flow Graph)
- Every variable name as a known value
- Explicit merging of values
- Easy to follow use-def links, O(1) time and size
- Easy to maintain def-uses chains
- Easy to rename a definition (move elimination)
- Difficult to add new variables, and thus new definitions: a new complete SSA construct pass must be performed for these new variables.

Program Intermediate Representations

- Several kind and level of Intermediate Representation (IR)
- Program IR can have SSA form property, thanks to:
  - Φ–nodes for control flow merges
  - KILL-nodes for enforcing reaching definition
  - Special nodes in case of predicated definitions (ref to PSI-SSA)
  - Additional info to track pre-allocated variables (ref to out of-SSA)
  - Other for new IR…
- Alternatives when not easy for a given IR:
  - Consider just a subset of variables
    - For instance: do not consider pre-allocated variables
  - Consider just a sub region of the program
    - For instance: SSA for basic block only

SSA construction

- There are multiple solutions to transform a program IR into SSA form. There are common ways.

- The set of nodes that need Φ–nodes for any variable V is the iterated dominance frontier $DF_+(L)$, where L is the set of nodes with assignments to V

- Semi-Pruned SSA: No Φ–nodes for local variables (There is always a def before a use in a basic block)

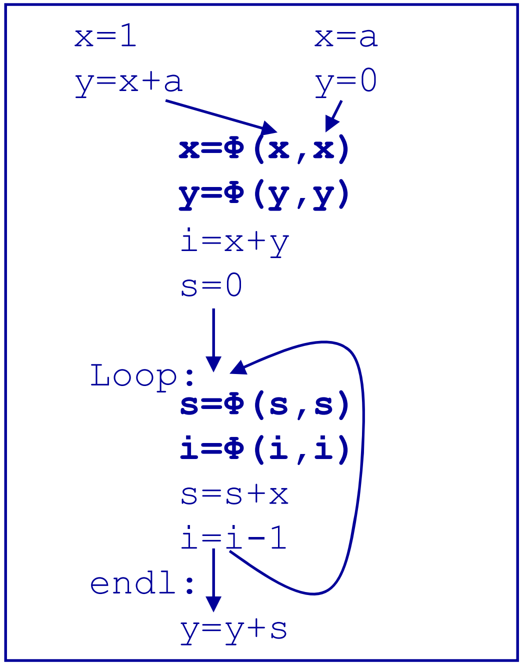
- Pruned SSA: Uses live-analysis to insert Φ–nodes only where they are live

SSA Construction

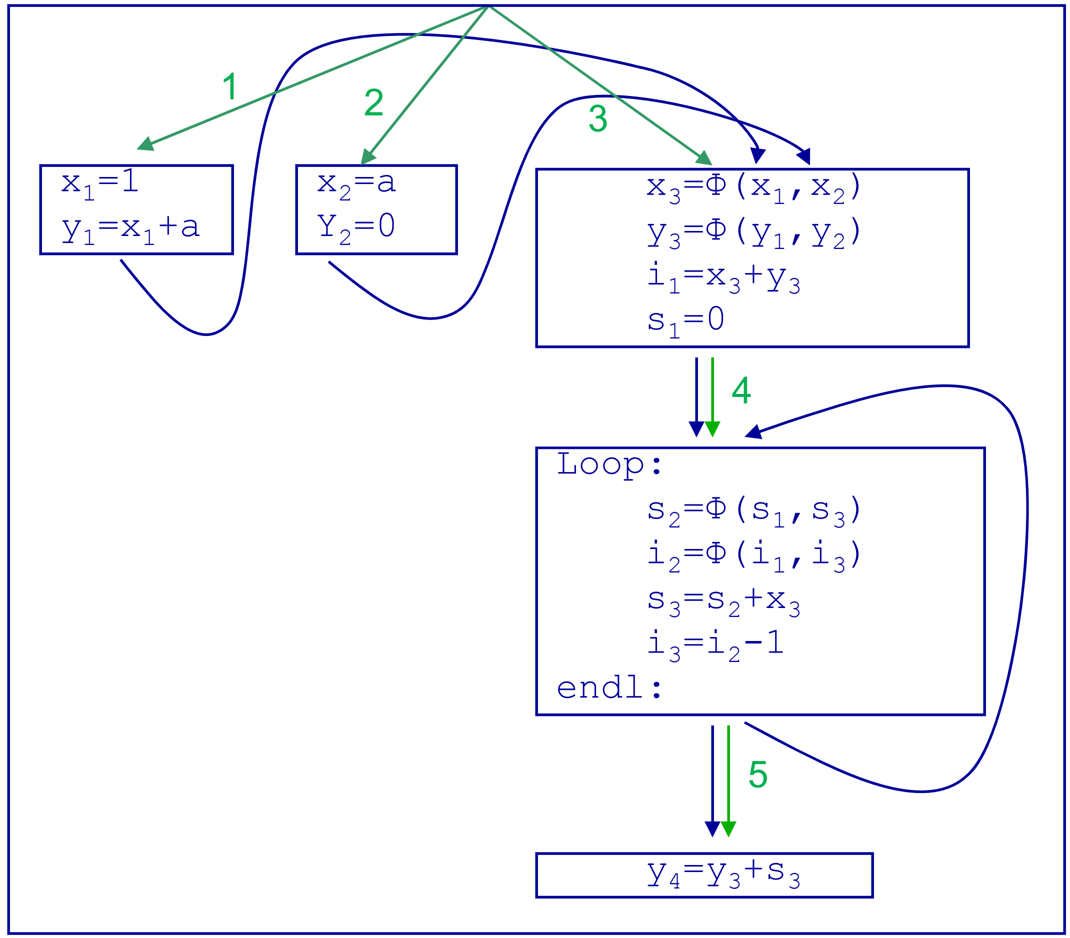
- For each variable V in the program
  - Find nodes where V is defined
  - Compute the iterated dominance frontier of these nodes
  - Place Φ–nodes on the iterated dominance frontier
- Rename the variable
  - Walk the dominator tree in preorder
  - Maintain a stack of renaming for each variable
  - Create new names on definitions, rename uses
  - Fill Φ–nodes arguments in successor nodes

SSA Contruction (Cont'd)

Φ-insertion



renaming



SSA Destruction: out of-SSA problem

- Issues:
  - Φ-nodes (pseudo ops) are not executable
  - pre-allocation constraints (pseudo args) must be explicited

- Out of-SSA problem:
  get a functionally equivalent program without pseudo ops or pseudo args

SSA Destruction: Example

- Out of-SSA example:
  - From the original C code we get the IR in SSA form
  - Some transformations have been performed
  - Out-of SSA transforms the SSA form program into the executable form.

```
int f(int a, int b)
int x=a+b;
if (x>0)
{
    a=b;
}


x=g(a)
a=x+a;
return a;
```

C code

```
R1_0,R2_0=pseudo_entry
x_1=R1_0+R2_0
b_1=(x_1>0)

R1_1=R2_0

R1_2= Φ(R1_1,R1_0)
x_2=pseudo_call g(R1_2)
x_3=x_2+R1_2
R16_3=X_3
pseuso_return R16_3
```

SSA form

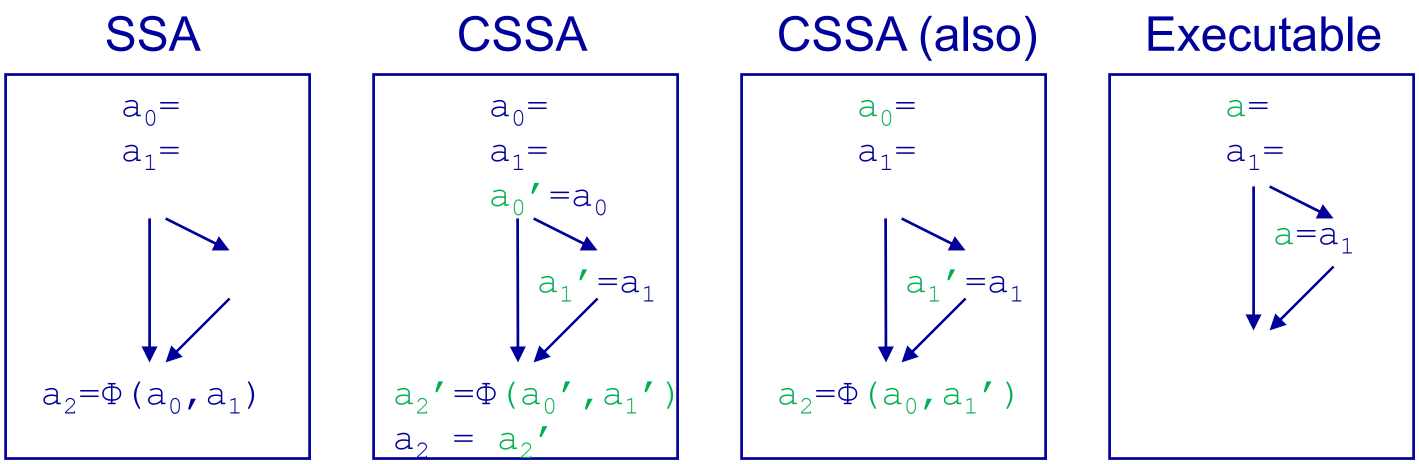
```
/* R1, R2 params */
x_1=R1+R2
b_1=(x_1>0)

R1=R2

R1_2 = R1
call g /* R1=g(R1) */
x_3=R1+R1_2
R1=X_3
return /* R1 */
```

Executable form

SSA Destruction: An approach

- Perform out of-SSA in two steps:
  - Convert to conventional SSA (CSSA)
  - Then perform renaming and discard pseudo ops
- Conventional SSA:
  - In this form, there is no interference between variables in a transitive closure of results and arguments of PHI operations
  - The result of the SSA construction, when no copy propagation is performed, is conventional
  - Most SSA transformations, such as copy propagation or code motion, may create non-conventional SSA

SSA Destruction: CSSA example



- There are several algorithms to convert SSA to Conventional SSA:
  - Some are wrong or do not account for IR specificities
  - Some trivial algorithms insert lot of copies
  - Advanced ones coalesce these copies or avoid their insertion
- Alternative: maintain CSSA form all along the IR
  - Very hard and bug prone: do not rely on this

Optimizations over SSA form programs

- Most standard algorithms have an SSA version, usually more efficient:
  - use-def-uses chains are maintained along transformations
  - Information on each variable is valid globally
  - Dominance property simplify algorithmic complexity

- Examples:
  - Copy Propagation: straight-forward during SSA renaming phase
  - Sparse Conditional Constant Propagation and other data flow analysis: use-def-uses chains and dominance property.
  - Dead-Code Elimination: mark side effects ops and recurse on use defs links
  - Detection of loop induction variables and determination of loop trip count is quite easy
  - Partial-redundancy elimination: still quite hard but more easy and efficient
  - Register allocation: chordal interference graph property makes coloring polynomial

The Psi-SSA form

- Psi-SSA form presentation outline:
  - Motivation
  - Definition
  - Properties
  - Benefits
  - Construction
  - Transformations
  - Destruction
- Conclusion

Motivation for Psi-SSA

- Motivation:
  - Enable SSA form property at machine code level
- Why?
  - Run all the efficient SSA based algorithms at this level (accuracy)
- One of the issues:
  - Target processors with full or partial support for predication
    - We can not statically determine anymore reaching definitions!
- Predicated instructions are “optional definitions”, example:

```
              ?
 p?  x =1  ⟶  execute x=1 only if p is true

!p?  x =2  ⟶  execute x=2 only if !p is true

    =x
```

Definition of Psi-SSA

- Psi-SSA **is** a SSA form (Single Static Assignment)
- Psi-SSA adds support for predicated instructions
  - introduce a new ψ pseudo-op to keep SSA property

SSA representation	Psi-SSA representation
if (p)	
$a_1 = 1$	$p?\ a_1 = 1$
else	
$a_2 = -1$	$!p?\ a_2 = -1$
$a_3 = \Phi(a_1, a_2)$	$a_3 = \psi(p?a_1, !p?a_2)$
if (q)	
$b_1 = 0$	$q?\ b_1 = 0$
$b_2 = \Phi(a_3, b_1)$	$b_2 = \psi(a_3, q?b_1)$
$b_3 = a_3 + b_2$	$b_3 = a_3 + b_2$

Properties of Psi-SSA

- A Psi operation merges values defined on different predicates
- Predicates on definitions are ignored
- The result of a Psi operation is a non-predicated definition
- The execution of a Psi operation returns the value of the rightmost variable whose predicate is true at runtime

$$
\begin{array}{lll}
 & a_1 & = 1 \\
p? & a_2 & = -1 \\
 & a_3 & = \psi(1?a_1, p?a_2) \\
q? & b_1 & = 0 \\
 & b_2 & = \psi(1?a_1, p?a_2, q?b_1) \\
 & b_3 & = a_3 + b_2
\end{array}
$$

Properties of Psi-SSA (cont'd)

- A predicate is associated with each argument
  - Allow for speculation of predicated definitions
  - Provide support for partial predication
- Predicate domains need not be disjoint
  - Several predicates may be true at the same time.
  - The order of the arguments in a Psi operation is significant

$$
\begin{array}{lll}
 & a_1 & = 1 \\
p? & a_2 & = -1 \\
 & a_3 & = \psi(1?a_1, p?a_2) \\
q? & b_1 & = 0 \\
 & b_2 & = \psi(1?a_1, p?a_2, q?b_1) \\
 & b_3 & = a_3 + b_2
\end{array}
$$

Benefits of Psi-SSA

- Easy to implement on top of an SSA representation
- No penalty if no predicated operation
- More flexibility in optimization ordering for predicated instruction sets
  - SSA algorithms are easy to adapt to the Psi-SSA representation (just add the support for the new pseudo op)
  - If-Conversion under SSA
- Specific optimizations on predicated code
  - Predicate promotion

Benefits of Psi-SSA (cont'd)

- Standard SSA algorithms can be used on Psi-SSA
  - Predicated instructions are treated as unconditional
  - New rules have to be defined on Psi operations
  - constant propagation, dead code elimination, global value numbering have been adapted to this representation

- Example: Constant propagation

```
      a1 = 1                  -> 1
  p?  a2 = a1+1               -> 2
 !p?  a3 = 2                  -> 2
      a4 = ψ(p?a2,!p?a3)      -> 2
```

Construction of Psi-SSA

- During the SSA construction
  - Insertion of Psi operation after predicated definitions

```
    a₁ = 0
p?  a₂ = 1
    a₃ = ψ(1?a₁,p?a₂)
```

- While in SSA form by an if-conversion algorithm
  - Transformation of Phi operations into Psi operations

```
a₁ = 0
if (p)
    a₂ = 1
a₃ = Φ(a₁,a₂)
```

```
    a₁ = 0

p?  a₂ = 1
    a₃ = ψ(1?a₁,p?a₂)
```

Transformations on Psi-SSA

- Some transformations on Psi operations:
  - Psi-inlining

```
x= ψ(p?a,q?b)
y= ψ(p|q?x,r?c)
            -> y= ψ(p?a,q?b,r?c)
```

  - Psi-reduction

```
x= ψ(p?a,q?b,p?c)
                  -> x= ψ(q?b,p?c)
```

  - Psi-projection

```
x= ψ(p?a,q?b) /* p∩q = ∅ */
                  -> x1= ψ(p?a)
p? z=x /* single use of x*/
                  -> p? z=x1 //z=a
```

  - Psi-promotion

```
x= ψ(p?a,q?b)
                  -> x= ψ(1?a,q?b)
```

Destruction of Psi-SSA

- Variables connected through a Psi operation must be renamed into a single variable, but:
  - Code motion may have changed the order in which predicated definitions occur
  - Operation speculation may have assigned a different predicate on a variable's definition and on its use in a Psi operation
  - Copy folding may have introduced interferences between variables in Psi operations

$$\begin{aligned} p?\ a_2 &= 1 \\ a_1 &= 0 \\ a_3 &= \psi(1?a_1, p?a_2) \end{aligned}$$

$$\begin{aligned} a_1 &= 0 \\ a_2 &= 1 \\ a_3 &= \psi(1?a_1, q?a_2) \end{aligned}$$

$$\begin{aligned} a_1 &= 0 \\ p?\ a_2 &= 1 \\ a_3 &= \psi(1?a_1, p?a_2) \\ q?\ b_2 &= -1 \\ b_3 &= \psi(1?a_3, q?b_2) \\ c_1 &= a_3 + b_3 \end{aligned}$$

Destruction of Psi-SSA (cont'd)

- Implemented as two steps above the out of-SSA algorithm

- A Psi-Normalize step
  - Restores the order of predicated definitions
  - Uses the same predicate on a variable's definition and on its use in a Psi operation
- A Psi-congruence step
  - Insert copies to remove interferences in psi-congruence classes
  - Uses a special definition for liveness on normalized Psi operations

```
p? a1 =
q? a2 =
   a3 =    ψ(p?a1,q?a2)
```

Destruction of Psi-SSA (cont'd)

- Predicated copies are generated to repair non-normalized Psi operations and interferences between Psi arguments
- Interferences between Psi arguments must also take into account interferences on Phi operations
- A simple Predicate Query System is used to eliminate false interferences on disjoint predicates

Conclusion

- Algorithms to build, optimize and deconstruct the Psi-SSA representation are well defined
- The Psi-SSA representation has proven to be a very effective representation to applying transformations on predicated code for our target processors
- Standard SSA algorithms are easy to adapt to Psi-SSA
- The Psi-SSA representation gives more flexibility in the ordering of optimizations in the compiler back-end

References

- Compilers using SSA
  - Middle-end: gcc, open64
  - Machine level: open64 (at ST), LAO (at ST), llvm, HotSpot
- Our contributions to SSA/Psi-SSA representation:
  - "Optimizing Translation Out of SSA Using Renaming Constraints"
    F. de Ferrière, C.Guillon, F.Rastello – CGO-2004
  - *"Revisiting Out of SSA Translation for Correctness, Efficiency and Speed"*
    B.Boissinot, A.Darte, B.Dupont de Dinechin, C.Guillon, F.Rastello – CGO-2009
  - "*Efficient static single assignment form for predication"*
    A.Stouchinin, F. de Ferrière - Micro-34
  - *"Improvements to the Psi-SSA Representation"*
    F. de Ferrière – Scopes 2007