Chapter 1

Static Single Information from a Functional Perspective

Jeremy Singer[1]

Abstract: Static single information form is a natural extension of the well-known static single assignment form. It is a program intermediate representation used in optimising compilers for imperative programming languages. In this paper we show how a program expressed in static single information can be transformed into an equivalent program in functional notation. We also examine the implications of this transformation.

1.1 INTRODUCTION

Static Single Information form (SSI) is a natural extension of the well-known Static Single Assignment form (SSA). SSA is a compiler intermediate representation that enables precise and efficient analyses and optimisations.

In SSA, each variable in the program has a unique definition point. In order to achieve this, it is necessary to rename variables, and insert extra pseudo-definitions (ϕ-functions) at control flow merge points.

We take the following simple program as an example:

```
1:  z ← input()
2:  if (z = 0)
3:      then y ← 42
4:      else y ← z + 1
5:  output(y)
```

[1]University of Cambridge Computer Laboratory,
William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
Email: jeremy.singer@cl.cam.ac.uk

ϕ-function	σ-function
inserted at control flow merge points	inserted at control flow split points
placed at start of basic block	placed at end of basic block
single destination operand	n destination operands, where n is the number of successors to the basic block that contains this σ-function
n source operands, where n is the number of predecessors to the basic block that contains this ϕ-function.	single source operand
takes the value of one of its source operands (dependent on control flow) and assigns this value to the destination operand	takes the value of its source operand and assigns this value to one of the destination operands (dependent on control flow)

FIGURE 1.1. Differences between ϕ- and σ-functions

To convert this program into SSA form, we have to rename instances of variable y so that each new variable has only a single definition point in the program.

The SSA version of the program is shown below:

$$
\begin{array}{ll}
1: & z \leftarrow \text{input}() \\
2: & \textbf{if } (z = 0) \\
3: & \qquad \textbf{then } y_0 \leftarrow 42 \\
4: & \qquad \textbf{else } y_1 \leftarrow z + 1 \\
5: & y_2 \leftarrow \phi(y_0, y_1) \\
6: & \text{output}(y_2)
\end{array}
$$

So we see that the ϕ-function merges (or multiplexes) the two incoming definitions of y_0 and y_1 at line 5. If the path of execution comes from the then branch, then the ϕ-function takes the value of y_0. Whereas if the path of execution comes from the else branch, then the ϕ-function takes the value of y_1.

SSI is a natural extension of SSA. It introduces another pseudo-definition, the σ-function. When converting to SSI form, in addition to renaming variables, and inserting ϕ-functions at control flow merge points, it is necessary to insert σ-functions at control flow split points.

The σ-function is the exact opposite of the ϕ-function. The differences are tabulated in figure 1.1.

We now convert the above program into SSI form:

$$
\begin{array}{ll}
1: & z_0 \leftarrow \text{input}() \\
2: & \textbf{if } (z_0 = 0) \\
3: & z_1, z_2 \leftarrow \sigma(z_0) \\
4: & \qquad \textbf{then } y_0 \leftarrow 42
\end{array}
$$

```
5:        else y1 ← z2 + 1
6:   y2 ← φ(y0, y1)
7:   output(y2)
```

So we see that the σ-function splits (or demultiplexes) the outgoing definition of z_0 at line 3. If the path of execution proceeds to the then branch, then the σ-function assigns the value of z_0 to z_1. However if the path of execution proceeds to the else branch, then the σ-function assigns the value of z_0 to z_2.

Since SSI is such a natural extension of SSA, it follows that algorithms for SSA can be quickly and naturally modified to handle SSI. For example the standard SSA construction algorithm [CFR+91] can be simply extended to construct SSI instead [Sin02]. Similarly, the SSA conditional constant propagation algorithm [WZ91] has a natural analogue in SSI [Ana99], which produces even better results.

It is well known that SSA can be considered as a form of functional programming [App98b]. Inside every SSA program, there is a functional program waiting to be released. Therefore, we should not be surprised to discover that SSI can also be considered as a form of functional programming.

For example, consider the following program, which calculates the factorial of 5.

```
1:  r ← 1
2:  x ← 5
3:  while (x > 0) do
4:      r ← r * x
5:      x ← x − 1
6:  done
7:  return r
```

First we convert this program into a standard control flow graph (CFG) [ASU86], as shown in figure 1.2.

Now we translate this program into SSI form, as shown in figure 1.3.

This SSI program can be simply transformed into the functional program shown in figure 1.4.

In the conversion from SSA to functional notation, a basic block that begins with a φ-function is transformed into a function. Jumps to such basic blocks become tail calls to the corresponding functions. The actual parameters of the tail calls are the source operands of the φ-functions. The formal parameters of the corresponding functions are the destination operands of the φ-functions.

In the conversion from SSI to functional notation, in addition to the above transformation, whenever a basic block ends with one or more σ-functions, then successor blocks are transformed into functions. Jumps to such successor blocks become tail calls to the corresponding functions. The actual parameters of the tail calls are the source operands of the σ-functions. The formal parameters of the corresponding functions are the relevant destination operands of the σ-functions. (We notice again that σ-functions have analogous properties to φ-functions.)

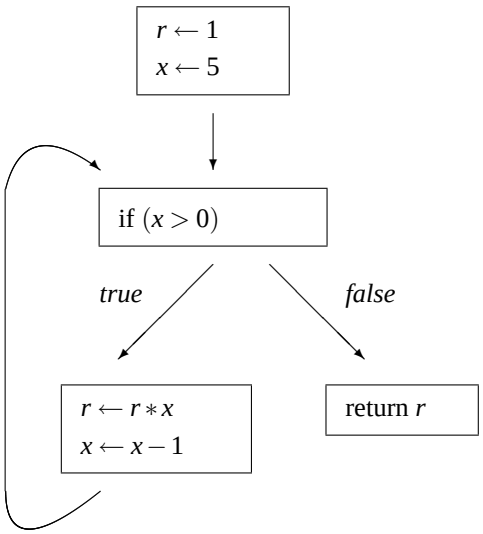

FIGURE 1.2. Control flow graph for factorial program

The remainder of this paper is laid out as follows: in section 1.2 we review the work already done in this area, in section 1.3 we formally define SSI, in section 1.4 we present the algorithm to transform SSI code into a functional program, in section 1.5 we show how there is both an optimistic and a pessimistic version of this transformation, in section 1.6 we discuss why this transformation may be useful, then finally in section 1.7 we draw some conclusions.

1.2 RELATED WORK

To the best of our knowledge no-one has attempted to transform SSI into a functional notation. Ananian [Ana99] gives an executable representation for SSI, but this is defined in terms of demand-driven operational semantics, and seems rather complicated.

Several people have noted a correspondence between programs in SSA and λ-calculus. Kelsey [Kel95] shows how to convert continuation passing style into SSA and vice versa.

Appel [App98b] informally shows the correspondence between SSA and functional programming. He gives an algorithm [App98a] for translating SSA to a functional intermediate form. (We extend Appel's algorithm in section 1.4 of this paper.)

Chakravarty et al [CKZ03] formalise a mapping from programs in SSA form

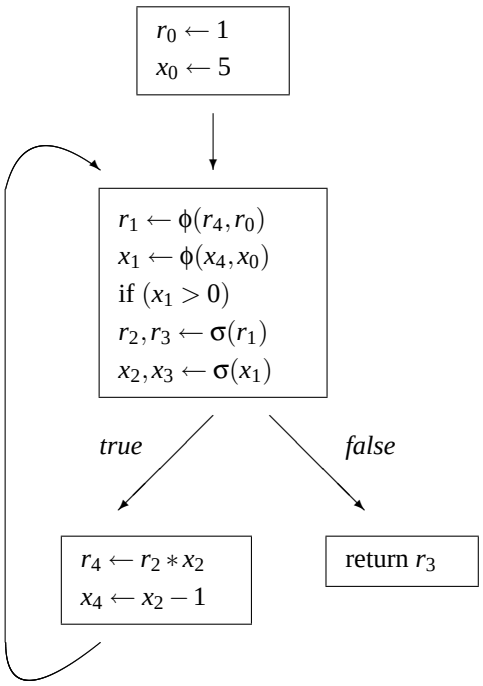

FIGURE 1.3. Static single information form for factorial program

to administrative normal form (ANF). ANF is a restricted form of λ-calculus. They also show how the standard SSA conditional constant propagation algorithm [WZ91] can be rephrased in terms of ANF programs.

1.3 STATIC SINGLE INFORMATION

Static Single Information form (SSI) was originally described by Ananian [Ana99]. He states that "the principal benefits of using SSI form are the ability to do predicated and backwards data flow analyses efficiently." He gives several examples including very busy expressions analysis and sparse predicated typed constant propagation. Indeed, SSI has been applied to a wide range of problems [RR00, SBA00, GSR03, AR03]

The MIT Flex compiler [Fle98] uses SSI as its intermediate representation. Flex is a compiler for Java, written in Java. As far as we are aware, Flex is the only publicly available SSI-based compiler. However, we are adding support for SSI to the Machine SUIF compiler infrastructure [Smi96]. We have implemented

```
let r0 = 1, x0 = 5
in
  let function f2(r1,x1) =
    let function f3(r2,x2) =
      let r4 = r2 * x2, x4 = x2 − 1
      in
        f2(r4,x4)
    and function f4(r3,x3) =
      return r3
    in
      if (x1 > 0)
        then f3(r1,x1)
        else f4(r1,x1)
  in
    f2(r0,x0)
```

FIGURE 1.4. Functional representation for SSI factorial program

an efficient algorithm for SSI construction [Sin02] and several new SSI analysis passes.

Below, we give the complete formal definition of a transformation from CFG to SSI notation. This definition is taken from Ananian [Ana99].

A few auxiliary definitions may be required before we quote Ananian's SSI definition. The original program is the classical CFG representation of the program [ASU86]. Program statements are contained within nodes. Directed edges between nodes represent the possible flow of control. A path is a sequence of consecutive edges. $\rightarrow^+$ represents a path consisting of at least one edge (a nonnull path). There is a path from the START node to every node in the control flow graph, and there is a path from every node in the control flow graph to the END node. The new program is in SSI. It is also a control flow graph, but it contains additional pseudo-assignment functions and the variables have been renamed. The variables in the original program are referred to as the original variables. The SSI variables in the new program are referred to as the new variables.

So, here is Ananian's definition:

1. If two nonnull paths $X \rightarrow^+ Z$ and $Y \rightarrow^+ Z$ exist having only the node Z where they converge in common, and nodes X and Y contain either assignments to a variable V in the original program or a ϕ- or σ-function for V in the new program, then a ϕ-function for V has been inserted at Z in the new program. (Placement of ϕ-functions)

2. If two nonnull paths $Z \rightarrow^+ X$ and $Z \rightarrow^+ Y$ exist having only the node Z where they diverge in common, and nodes X and Y contain either uses of a variable V in the original program or a ϕ- or σ-function for V in the new program, then

a σ-function for V has been inserted at Z in the new program. (Placement of σ-functions)

3. For every node X containing a definition of a variable V in the new program and node Y containing a use of that variable, there exists at least one path $X \rightarrow^{+} Y$ and no such path contains a definition of V other than at X. (Naming after ϕ-functions)

4. For every pair of nodes X and Y containing uses of a variable defined at node Z in the new program, either every path $Z \rightarrow^{+} X$ must contain Y or every path $Z \rightarrow^{+} Y$ must contain X. (Naming after σ-functions)

5. For the purposes of this definition, the START node is assumed to contain a definition and the END node a use for every variable in the original program. (Boundary conditions)

6. Along any possible control flow path in a program being executed consider any use of a variable V in the original program and the corresponding use V_i in the new program. Then, at every occurrence of the use on the path, V and V_i have the same value. The path need not be cycle-free. (Correctness)

Construction of SSI can be performed in $O(EV)$ time, where E is a measure of the number of edges in the control flow graph and V is a measure of the number of variables in the original program. This is worst case complexity, but typical time complexity is linear in the program size.

1.4 TRANSFORMATION

In this section we present the algorithm that transforms from SSI into a functional notation.

We will adopt a cut-down version of Appel's functional intermediate representation [App98a]. The abstract syntax of our functional notation is given in figure 1.5.

Expressions are broken down into primitive operations whose order of evaluation is specified. Every intermediate result is an explicitly named temporary. Every argument of an operator or function is an atom (variable or constant). As in SSA, SSI and λ-calculus, every variable has a single assignment (binding), and every use of that variable is within the scope of the binding. (In figure 1.5, binding occurrences of variables are underlined.) No variable name can be used in more than one binding. Every binding of a variable has a scope within which all the uses of that variable must occur.

- For a variable bound by **let** $v = \ldots$ **in** *exp*, the scope of *v* is just *exp*.
- The scope of a function variable f_i bound in

$$\begin{array}{l} \textbf{let function } f_1(\ldots) = exp_1 \ldots \\ \quad \textbf{function } f_k(\ldots) = exp_k \end{array}$$

atom	→ *c*	constant integer
atom	→ *v*	variable
exp	→ **let** *fundefs* **in** *exp*	function declaration
exp	→ **let** $\underline{v}$ = *atom* **in** *exp*	copy
exp	→ **let** $\underline{v}$ = *binop*(*atom*,*atom*) **in** *exp*	arithmetic operator
exp	→ **if** *atom relop atom* **then** *exp* **else** *exp*	conditional branch
exp	→ *atom*(*args*)	tail call
exp	→ **let** $\underline{v}$ = *atom*(*args*) **in** *exp*	non-tail call
exp	→ **return** *atom*	return
args	→	
args	→ *atom args*	
fundefs	→	
fundefs	→ *fundefs* **function** $\underline{v}$(*formals*) = *exp*	
formals	→	
formals	→ $\underline{v}$ *formals*	
binop	→ **plus** | **minus** | **mul** | ...	
relop	→ **eq** | **ne** | **lt** | ...	

FIGURE 1.5. Functional intermediate representation

in *exp*

includes all the exp_j (to allow for mutually recursive functions) as well as *exp*.

- For a variable bound as the formal parameter of a function, the scope is the body of that function.

Any SSI program can be translated into this functional form. Each basic block with more than one predecessor is transformed into a function. The formal parameters of that function are the destination operands of the ϕ-functions in that basic block. Similarly, each basic block which is the target of a conditional branch instruction is transformed into a function. The formal parameters of that function are the appropriate destination operands of the σ-functions in the preceding basic block (that is to say, the σ-functions that are associated with the conditional branch). We assume that the SSI program is in edge-split form—no basic block with multiple successors has an edge to a basic block with multiple predecessors. In particular this means that blocks which are the targets of a conditional branch can only have a single predecessor. (It should always be possible to transform an SSI program into edge-split form.)

If block f dominates block g, then the function for g will be nested inside the body of the function for f. Instead of jumping to a block which has been transformed into a function, a tail call replaces the jump. The actual parameters of the tail call will be the appropriate source operands of corresponding σ- or ϕ-functions. (Notice that every conditional branch will dominate both its then and else blocks, in edge-split SSI.)

The algorithm for transforming SSI into functional intermediate form is based on algorithm 19.20 from Appel's book [App98a]. Appel's algorithm handles SSA, so we extend it to deal with SSI instead.

In the algorithm (figure 1.6) lines of code that have been altered from Appel's original SSA-based algorithm are marked with a ! and entirely new lines of code (to handle SSI-specific cases) are marked with a +

1.5 COMPUTING SSI

There are two different approaches to constructing SSI form. Ananian's approach [Ana99] is pessimistic—it assumes that ϕ- and σ-functions are needed everywhere, and then it removes such functions when it can show that they are not actually required. This is a kind of greatest fixed point calculation. The alternative approach [Sin02] is optimistic—it assumes that no ϕ- or σ-functions are needed, then it inserts such functions when it can show that they are actually required. This is a kind of least fixed point calculation. (Ananian claims that this optimistic approach ought to take longer, but in practice it seems to be more efficient than the pessimistic approach.)

Just as there is an optimistic and a pessimistic approach to the computation of SSI, there appear to be an optimistic and a pessimistic approach to the transformation into functional notation. The pessimistic approach takes the original program control flow graph (not in SSI form) and converts each basic block into a top-level function, with tail calls to appropriate successor functions. It then applies standard lambda lifting techniques [Joh85] to generate the appropriate parameters for each functional block.

The optimistic approach is exactly as given in section 1.4. It uses the dominance relations of the control flow graph to determine how the functional blocks should be nested. Then it (effectively) does a restricted amount of lambda lifting to generate the appropriate parameters for each function.

A formal clarification of the relationship between optimistic and pessimistic construction of SSI is the subject of ongoing research.

1.6 USES

In this section we briefly consider why the transformation from SSI into functional notation may be of value.

Typed functional languages may be useful as intermediate representations in compilers for imperative languages. It is certainly true that algorithms on such functional representations can often be more rigorously defined [CKZ03]. It is

interesting to compare existing SSA or SSI data flow analyses with the equivalent analyses in the functional paradigm, perhaps to discover similarities and differences. Such cross-community experience is often instructive to one of the parties, if not both.

We have effectively made SSI interprocedural in scope, by abstracting all control flow into function calls. Until now, SSI has only been envisaged as an intraprocedural representation, and it has not been clear how to extend SSI to whole program scope.

Finally we note that the functional representation of SSI programs is executable. Standard SSI is not an executable representation, it is restricted in the same manner as original SSA. Ananian has concocted an operational semantics for an extended version of SSI [Ana99], however this is quite complex and unwieldy to use. On the other hand, functional programs are natural, understandable and easily executable with a well-known semantics. We have successfully translated some simple SSI programs into Haskell and ML code, using the transformation algorithm of section 1.4.

1.7 CONCLUSIONS

In this paper we have shown how SSI (generally regarded as an imperative program representation) can be converted into a simple functional notation. We have specified a tranformation algorithm and we have briefly discussed the possible applications of this transformation process.

Compilers for functional programming languages (such as the Glasgow Haskell compiler) often translate their intermediate form into an imperative language (such as C) which is then compiled to machine code. We are proposing to use a functional notation as the intermediate form in an imperative language compiler, which seems to be going against the current trend.

Finally we comment on future work. The transformation algorithm presented in section 1.4 could possibly be formalised, in the same manner as Appel's original work on SSA [App98b, App98a] has been formalised [CKZ03]. Next we need to translate existing SSI analysis algorithms to this new functional framework. We must also consider how to take advantage of this functional notation in order to devise new analyses and optimisations.

REFERENCES

[Ana99] C. Scott Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Sep 1999.

[App98a] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, first edition, 1998.

[App98b] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr 1998.

[AR03] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*, 2003. To appear.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[CKZ03] Manuel M.T. Chatravarty, Gabriele Keller, and Patryk Zadarnowski. A functional perspective on SSA optimisation algorithms. In *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification*, 2003. To appear.

[Fle98] The Flex compiler infrastructure, 1998. `http://www.flex-compiler.lcs.mit.edu/Harpoon/`.

[GSR03] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pages 273–284, Jan 2003.

[Joh85] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, Mar 1995.

[RR00] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 108–120, 2000.

[Sin02] Jeremy Singer. Efficiently computing the static single information form, 2002. `http://www.cl.cam.ac.uk/~jds31/research/computing.pdf`.

[Smi96] Michael D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, Jan 1996. `http://www.eecs.harvard.edu/machsuif/`.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr 1991.

```
 1: Translate(node) =
 2:     let C be the children of node in the dominator tree
 3:     let p_1, ..., p_n be the nodes of C that have more than one predecessor
 4:     for i ← 1 to n
 5:         let a_1, ..., a_k be the targets of φ-functions in p_i (possibly k = 0)
 6:         let S_i = Translate(p_i)
 7:         let F_i = "function f_{p_i}(a_1, ..., a_k) = S_i"
+8:     let s_1, ..., s_m be the nodes of C that are the target of a conditional branch
+9:     for i ← 1 to m
+10:        let q_i be the (unique) predecessor of s_i
+11:        let a_1, ..., a_k be the targets (associated with s_i) of σ-functions in q_i
+12:        let T_i = Translate(s_i)
+13:        let G_i = "function f_{s_i}(a_1, ..., a_k) = T_i"
!14:    let F = F_1 F_2 ... F_n G_1 G_2 ... G_m
 15:    return Statements(node, 1, F)

 16: Statements(node, j, F) =
 17:    if there are < j statements in node
 18:    then let s be the successor of node
 19:        if s has only one predecessor
 20:        then return Statements(s, 1, F)
 21:        else s has m predecessors
 22:            suppose node is the ith predecessor of s
 23:            suppose the φ-functions in s are
                    a_1 ← φ(a_1 1, ..., a_1 m), ...
                    a_k ← φ(a_k 1, ..., a_k m)
 24:            return "let F in f_s(a_1 i, ..., a_k i)"
 25:    else if the jth statement of node is a φ-function
 26:        then return Statements(node, j + 1, F)
+27:    else if the jth statement of node is a σ-function
+28:        then return Statements(node, j + 1, F)
 29:    else if the jth statement of node is "return a"
 30:        then return "let F in return a"
 31:    else if the jth statement of node is a ← b ⊕ c
 32:        then let S = Statements(node, j + 1, F)
 33:            return "let a = b ⊕ c in S"
 34:    else if the jth statement of node is a ← b
 35:        then let S = Statements(node, j + 1, F)
 36:            return "let a = b in S"
 37:    else if the jth statement of node is "if a < b then goto s_1 else goto s_2"
 38:        then since this is edge-split SSI form
 39:        assume s_1 and s_2 each has only one predecessor
!40:        let a_1, ..., a_k be
!               the source operands of σ-functions in node (possibly k = 0)
!41:        return "let F in if a < b then f_{s_1}(a_1, ..., a_k) else f_{s_2}(a_1, ..., a_k)
```

FIGURE 1.6. Algorithm that transforms from SSI to functional representation