Chapter 5

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 form can be transformed into an equivalent program in functional notation. We also examine the implications of this transformation.

5.1 INTRODUCTION

Static single information form (SSI) [2] is a natural extension of the well-known static single assignment form (SSA) [11]. SSA is a compiler intermediate representation for imperative programs that enables precise and efficient analyses and optimisations.

In SSA, each program variable has a unique definition point. To achieve this, it is necessary to rename variables and insert extra pseudo-definitions (ϕ-functions) at control flow merge points. Control flow merge points occur at the start of basic blocks. A basic block is a (not necessarily maximal) sequence of primitive instructions with the property that if control reaches the first instruction, then all instructions in the basic block will be executed. SSA programs have the desirable property of referential transparency—that is, the value of an expression depends only on the value of its subexpressions and not on the order of evaluation or side-effects of other expressions. Referentially transparent programs are easier to analyse and reason about.

We take the following simple program as an example:

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

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

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:

```
1:  $z \leftarrow$ input()
2:  **if** $(z = 0)$
3:      **then** $y_0 \leftarrow 42$
4:      **else** $y_1 \leftarrow z + 1$
5:  $y_2 \leftarrow \phi(y_0, y_1)$
6:  output($y_2$)
```

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 an extension of SSA. It introduces another pseudo-definition, the σ-function. When converting to SSI, 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. We have contrasted SSA and SSI at length elsewhere [25], in terms of their computation and data flow analysis properties. It is sufficient to say that SSI can be computed almost as efficiently as SSA and that SSI permits a wider range of data flow analysis techniques than SSA.

The σ-function is the exact opposite of the ϕ-function. The differences are tabulated in Fig. 5.1.

We now convert the above program into SSI:

```
1:  $z_0 \leftarrow$ input()
2:  **if** $(z_0 = 0)$
3:  $z_1, z_2 \leftarrow \sigma(z_0)$
4:      **then** $y_0 \leftarrow 42$
5:      **else** $y_1 \leftarrow z_2 + 1$
6:  $y_2 \leftarrow \phi(y_0, y_1)$
7:  output($y_2$)
```

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 straightforward extension of SSA, it follows that algorithms for SSA can be quickly and naturally modified to handle SSI. For example,

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 5.1 Differences between ϕ- and σ-functions

the standard SSA computation algorithm [11] can be simply extended to compute SSI instead [23]. Similarly, the SSA conditional constant propagation algorithm [29] has a natural analogue in SSI [2], which produces even better results.

It is a well-known fact that SSA can be seen as a form of functional programming [6]. 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 seen as a form of functional programming.

Consider the following program, which calculates the factorial of 5.

```
1:  $r \leftarrow 1$
2:  $x \leftarrow 5$
3:  **while** $(x > 0)$ **do**
4:      $r \leftarrow r * x$
5:      $x \leftarrow x - 1$
6:  **done**
7:  **return** $r$
```

First we convert this program into a standard control flow graph (CFG) [1], as shown in Fig. 5.2. Then we translate this program into SSI form, as shown in Fig. 5.3. This SSI program can be simply transformed into the functional program shown in Fig. 5.4.

In the conversion from SSA to functional notation, a basic block #n that begins with one or more ϕ-functions is transformed into a function f_n. 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

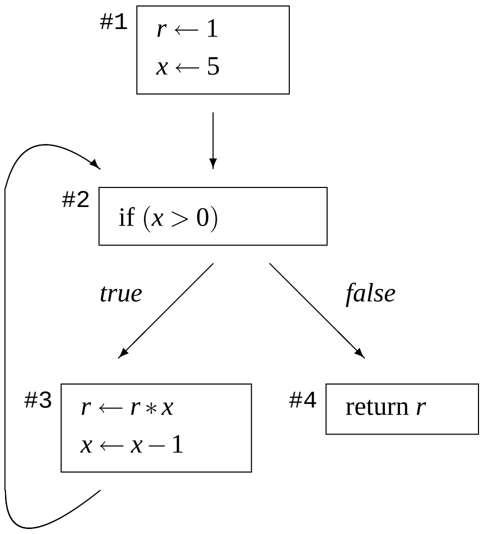

FIGURE 5.2 Control flow graph for factorial program

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 #p and #q are transformed into functions f_p and f_q. 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.)

The main technical contribution of this paper is the detailed presentation of an algorithm to convert SSI programs into a functional intermediate representation. The remainder of the paper is laid out as follows: in section 5.2 we review the previous work in this area, in section 5.3 we formally define SSI, in section 5.4 we present the algorithm to transform SSI code into a functional program, in section 5.5 we show how there are both an optimistic and a pessimistic version of this transformation, in section 5.6 we contemplate the possibility of recovering SSI from a functional program (the reverse transformation), in section 5.7 we discuss why the transformation from SSI to functional notation may be useful, then finally in section 5.8 we draw some conclusions.

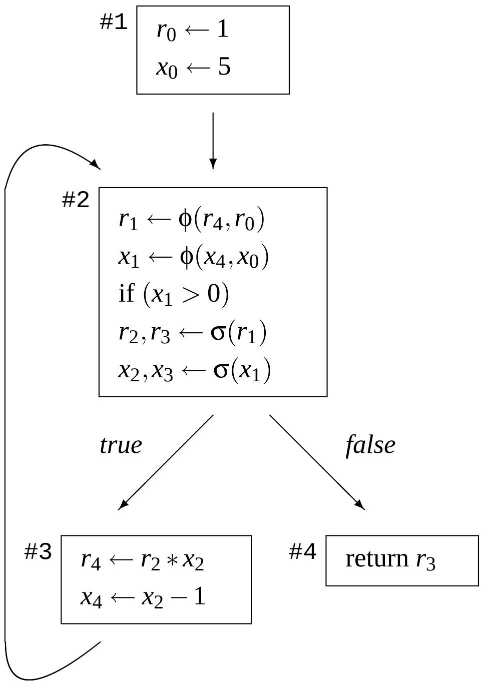
#1
$r_0 \leftarrow 1$
$x_0 \leftarrow 5$
#2
$r_1 \leftarrow \phi(r_4, r_0)$
$x_1 \leftarrow \phi(x_4, x_0)$
if $(x_1 > 0)$
$r_2, r_3 \leftarrow \sigma(r_1)$
$x_2, x_3 \leftarrow \sigma(x_1)$
true
false
#3
$r_4 \leftarrow r_2 * x_2$
$x_4 \leftarrow x_2 - 1$
#4
return r_3

FIGURE 5.3 Static single information form for factorial program

5.2 RELATED WORK

To the best of our knowledge no-one has attempted to transform SSI into a functional notation. Ananian [2] 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 [16] shows how to convert continuation passing style [4] into SSA and vice versa. Appel [6] informally shows the correspondence between SSA and functional programming. He gives an algorithm [5] for translating SSA to functional intermediate representation. (We extend Appel's algorithm in section 5.4 of this paper.)

Chakravarty et al. [10] formalise a mapping from programs in SSA form to administrative normal form (ANF) [12]. ANF is a restricted form of λ-calculus. They also show how the standard SSA conditional constant propagation algorithm [29] can be rephrased in terms of ANF programs.

```
let $r_0 = 1,\ x_0 = 5$
in
  let function $f_2(r_1, x_1) =$
    let function $f_3(r_2, x_2) =$
      let $r_4 = r_2 * x_2,\ x_4 = x_2 - 1$
      in
        $f_2(r_4, x_4)$
    and function $f_4(r_3, x_3) =$
      return $r_3$
    in
      if $(x_1 > 0)$
        then $f_3(r_1, x_1)$
        else $f_4(r_1, x_1)$
  in
    $f_2(r_0, x_0)$
```

FIGURE 5.4 Functional representation for SSI factorial program

5.3 STATIC SINGLE INFORMATION

Static single information form (SSI) was originally described by Ananian [2]. 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 [22, 28, 14, 3].

The MIT Flex compiler [13] 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 Machine SUIF [27], an extensible compiler infrastructure for imperative languages like C and Fortran. We have implemented an efficient algorithm for SSI computation [23] 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 [2]. 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 [1]. Program statements are contained within nodes (also known as basic blocks). 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 non-null path). There is a path from the START node to every node in the CFG and there is a path from every node in the CFG to the END node. The new program is in SSI. It is also a CFG, but it contains additional pseudo-definition 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)

Ananian's original SSI computation algorithm 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.

5.4 TRANSFORMATION

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

We adopt a cut-down version of Appel's functional intermediate representation (FIR) [5]. The abstract syntax of our FIR is given in Fig. 5.5. FIR has the same expressive power as ANF [12]. 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	$\rightarrow$ *c*	constant integer
atom	$\rightarrow$ *v*	variable
exp	$\rightarrow$ **let** *fundefs* **in** *exp*	function declaration
exp	$\rightarrow$ **let** $\underline{v}$ = *atom* **in** *exp*	copy
exp	$\rightarrow$ **let** $\underline{v}$ = *binop*(*atom*,*atom*) **in** *exp*	arithmetic operator
exp	$\rightarrow$ **if** *atom relop atom* **then** *exp* **else** *exp*	conditional branch
exp	$\rightarrow$ *atom*(*args*)	tail call
exp	$\rightarrow$ **let** $\underline{v}$ = *atom*(*args*) **in** *exp*	non-tail call
exp	$\rightarrow$ **return** *atom*	return
args	$\rightarrow$	
args	$\rightarrow$ *atom args*	
fundefs	$\rightarrow$	
fundefs	$\rightarrow$ *fundefs* **function** $\underline{v}$(*formals*) = *exp*	
formals	$\rightarrow$	
formals	$\rightarrow$ $\underline{v}$ *formals*	
binop	$\rightarrow$ **plus** | **minus** | **mul** | ...	
relop	$\rightarrow$ **eq** | **ne** | **lt** | ...	

FIGURE 5.5 Functional intermediate representation

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 Fig. 5.5, 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

  $$\textbf{let function } f_1(\ldots) = exp_1 \ldots \\ \quad \textbf{function } f_k(\ldots) = exp_k \\ \textbf{in } exp$$

  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 FIR. 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. (If the block has no ϕ-functions then it is transformed into a parameterless function.) Similarly, each basic block that 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 basic blocks that 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. (Every conditional branch will dominate both its `then` and `else` blocks, in edge-split SSI.)

The algorithm for transforming SSI into FIR is given in Fig. 5.6. It is based on algorithm 19.20 from Appel's book [5]. `Translate()` ensures function definitions are correctly nested. `Statements()` outputs FIR code for each basic block. Appel's algorithm handles SSA, so we extend it to deal with SSI instead. In our algorithm 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 `+`. In the code for the `Statements()` function, $\oplus$ represents the general case for binary arithmetic operators and $<$ represents the general case for binary relational operators.

5.5 OPTIMISTIC VERSUS PESSIMISTIC

There are two different approaches to computing SSI. Ananian's approach [2] is pessimistic, in that it assumes that ϕ- and σ-functions are needed everywhere, then it removes such functions when it can show that they are not actually required. This is a kind of greatest fixed point calculation. (Aycock and Horspool adopt the same pessimistic approach in their generation of SSA [8].) The alternative approach to computing SSI [23] 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. (The classical SSA computation algorithm [11] employs the same optimistic approach.) Ananian claims that this optimistic approach ought to take longer, but in practice it seems to be faster 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 CFG and converts each basic block into a top-level function, with tail calls to appropriate successor functions. Each generated top-level function has a formal parameter for every program variable, and each function call site has an actual parameter for every program variable. Appel [6] refers to this as the "really crude approach." Useless parameters may be identified and eliminated with the help of liveness and other data flow information. The necessary parameters for each functional block should be those variables which are live at each corresponding basic block boundary in the original program. (A variable is live at a particular program point if there is a control flow path from that point along which the variable's value may be used before that variable is redefined.) This makes sense since SSI is an encoding of liveness information, as Ananian states [2].

The optimistic approach is exactly as given in section 5.4. It can be explained in the following manner. It uses the dominance relations of the control flow graph to determine how the functional blocks should be nested. (Nesting is required in order for functional blocks to use variables declared in outer scope.) Then it applies standard lambda lifting techniques [15] to generate the appropriate parameters for each functional block.

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

5.6 CONVERTING FUNCTIONAL PROGRAMS BACK TO SSI

It is possible to transform an arbitrary program p expressed in FIR into SSI, simply by treating p as an imperative program. (Let-bound atomic variables become mutable virtual registers and function applications become procedure calls.) Standard SSI computation techniques [2, 23] can then be applied to the imperative program.

However, suppose that a program p_{SSI} in SSI has been transformed into a program p_{func} in FIR. In this section we address the concept of recovering p_{SSI} from p_{func}.

p_{func} is in SSA, since each let-bound variable is only assigned a value at one program point. However p_{func} is not in SSI, since the same parameters are supplied to the tail calls on either side of an `if` statement. (Recall that these parameters correspond to the source parameters of the σ-functions associated with this conditional branch in p_{SSI}.) The simplest way to transform p_{func} into a valid SSI program, say p'_{SSI}, is to add σ-functions at each `if` statement, and rename the parameters of the tail calls accordingly. There is a drawback with this approach however. Now imagine converting p'_{SSI} into FIR using our algorithm. There would be an additional layer of function calls at the `if` statements, because of the extra σ-functions. Admittedly these extra function calls could be removed by limited β-contraction, but it is embarrassing to admit that converting from SSI to FIR and back to SSI (ad infinitum) does not reach a fixed point. In fact this is a diverging computation.

The problem is that the σ-functions are already encoded as function calls in p_{func} but we do not recover this information. We insert extra σ-functions instead. One way to avoid this would be to inline (β-contract) all functions in p_{func} that are only called from one call site (this includes all functions that originated from σ-functions). If this transformation is done prior to the insertion of σ-functions, then the problem of an extra layer of function call indirection does not arise.

Kelsey [16] gives a method for recovering ϕ-functions from functional programs. We should be able to apply similar techniques to p_{func}. Thus it should be possible to recover (something resembling) p_{SSI} from p_{func}.

5.7 MOTIVATION

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 compiler intermediate representations for imperative languages. There has recently been a great deal of research effort in this area, with systems such as typed assembly language [18], proof carrying code [20, 7] and the value dependence graph [30]. SSA and SSI fit neatly into this category, since they can be seen from a functional perspective, and they are most amenable to high-level type inference techniques [19, 26]. The implementors of similar typed functional representations for Java bytecode, such as λJVM [17] and GRAIL [9], comment that a functional representation makes both verification and analysis straightforward. It is useful for reasoning about program properties, such as security and resource consumption guarantees. Functional notations are also well-suited for translation into lower-level program representations.

It is certainly true that algorithms on such functional representations can often be more rigorously defined [10] and proved correct. It would be 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. Now there is no longer any distinction between intraprocedural and interprocedural control flow.

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. (ϕ- and σ-functions require some kind of runtime support to determine which value to assign to which variable.) Ananian has concocted an operational semantics for an extended version of SSI [2], 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 5.4.

For instance, Fig. 5.7 shows the dynamic data flow graph [21] of three Haskell factorial functions that each compute 5! (the answer is 120). The three values are then added together (the sum total is 360). The left portion of the graph represents a standard Haskell iterative definition of the factorial function:

```
faci 0 acc = acc
faci n acc = fac1 (n-1) (acc*n)
```

The middle portion of the graph represents a standard Haskell recursive definition of the factorial function:

```
facr 0 = 1
facr n = n * facr (n-1)
```

The right portion of the graph represents the Haskell version of the functional program from figure 5.4 which is the transformation of the SSI program from figure 5.3. We notice that the right portion of the dynamic data flow graph has exactly the same shape as the left portion, which reveals that both are computing factorials iteratively, so we see that the transformation from imperative to functional style does not alter the data flow behaviour of the program at all.

5.8 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 transformation 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. This seems rather wasteful, since the C compiler (if it uses a functional representation as its intermediate form) will attempt to reconstruct the functional program which has been carelessly thrown away by the functional compiler backend.

Finally we comment on future work. The transformation algorithm presented in section 5.4 could possibly be formalised, in the same manner as Appel's original work on SSA [6, 5] has been formalised [10]. 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.

On a different note, SSA and SSI are just two members of a large family of renaming schemes [24]. It would be interesting to see if every scheme in the family could be converted to a functional notation, using the same general techniques outlined in this paper.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

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

[3] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, pages 59–68, 2003.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

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

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

[7] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 247–256, 2001.

[8] J. Aycock and N. Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer-Verlag, 2000.

[9] L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. *Electronic Notes in Theoretical Computer Science*, 85(1), 2003.

[10] M. M. Chatravarty, G. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. In *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification*, 2003. http://www.cse.unsw.edu.au/˜patrykz/papers/ssa-lambda/.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. 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.

[12] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

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

[14] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, 2003.

[15] T. 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.

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

[17] C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proceedings of the 5th World Conference on Systemics, Cybernetics, and Informatics—Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, 2001. http://flint.cs.yale.edu/flint/publications/lamjvm.html.

[18] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[19] A. Mycroft. Type-based decompilation. In *Proceedings of the European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer-Verlag, 1999.

[20] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[21] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[22] R. Rugina and M. 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.

[23] J. Singer. Efficiently computing the static single information form, 2002. http://www.cl.cam.ac.uk/˜jds31/research/computing.pdf.

[24] J. Singer. A framework for virtual register renaming schemes, 2003. http://www.cl.cam.ac.uk/˜jds31/research/renaming.pdf.

[25] J. Singer. SSI extends SSA. In *Work in Progress Session Proceedings of the Twelth International Conference on Parallel Architectures and Compilation Techniques*, 2003. http://www.cl.cam.ac.uk/˜jds31/research/ssavssi.pdf.

[26] J. Singer. Static single information improves type-based decompilation, 2003. http://www.cl.cam.ac.uk/˜jds31/research/ssidecomp.pdf.

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

[28] M. Stephenson, J. Babb, and S. 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.

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

[30] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, 1994.

```
  1: Translate($node$) =
  2: let $C$ be the children of $node$ in the dominator tree
  3: let $p_1, \ldots, p_n$ be the nodes of $C$ that have more than one predecessor
  4: for $i \leftarrow 1$ to $n$
  5:     let $a_1, \ldots, a_k$ be the targets of $\phi$-functions in $p_i$ (possibly $k = 0$)
  6:     let $S_i$ = Translate($p_i$)
  7:     let $F_i$ = "function $f_{p_i}(a_1, \ldots, a_k) = S_i$"
+ 8: let $s_1, \ldots, s_m$ be the nodes of $C$ that are the target of a conditional branch
+ 9: for $i \leftarrow 1$ to $m$
+ 10:    let $q_i$ be the (unique) predecessor of $s_i$
+ 11:    let $a_1, \ldots, a_k$ be the targets (associated with $s_i$) of $\sigma$-functions in $q_i$
+ 12:    let $T_i$ = Translate($s_i$)
+ 13:    let $G_i$ = "function $f_{s_i}(a_1, \ldots, a_k) = T_i$"
! 14: let $F = F_1 F_2 \ldots F_n G_1 G_2 \ldots G_m$
  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 $i$th predecessor of $s$
  23:         suppose the $\phi$-functions in $s$ are
                $a_1 \leftarrow \phi(a_1 1, \ldots, a_1 m), \ldots$
                $a_k \leftarrow \phi(a_k 1, \ldots, a_k m)$
  24:         return "let $F$ in $f_s(a_1 i, \ldots, a_k i)$"
  25: else if the $j$th statement of $node$ is a $\phi$-function
  26:     then return Statements($node, j+1, F$)
+ 27: else if the $j$th statement of $node$ is a $\sigma$-function
+ 28:     then return Statements($node, j+1, F$)
  29: else if the $j$th statement of $node$ is "return $a$"
  30:     then return "let $F$ in return $a$"
  31: else if the $j$th statement of $node$ is $a \leftarrow b \oplus c$
  32:     then let $S$ = Statements($node, j+1, F$)
  33:         return "let $a = b \oplus c$ in $S$"
  34: else if the $j$th statement of $node$ is $a \leftarrow b$
  35:     then let $S$ = Statements($node, j+1, F$)
  36:         return "let $a = b$ in $S$"
  37: else if the $j$th 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 $a_1, \ldots, a_k$ be
!             the source operands of $\sigma$-functions in $node$ (possibly $k = 0$)
! 41:     return "let $F$ in if $a < b$ then $f_{s_1}(a_1, \ldots, a_k)$ else $f_{s_2}(a_1, \ldots, a_k)$"
```

FIGURE 5.6 Algorithm that transforms SSI to functional intermediate representation

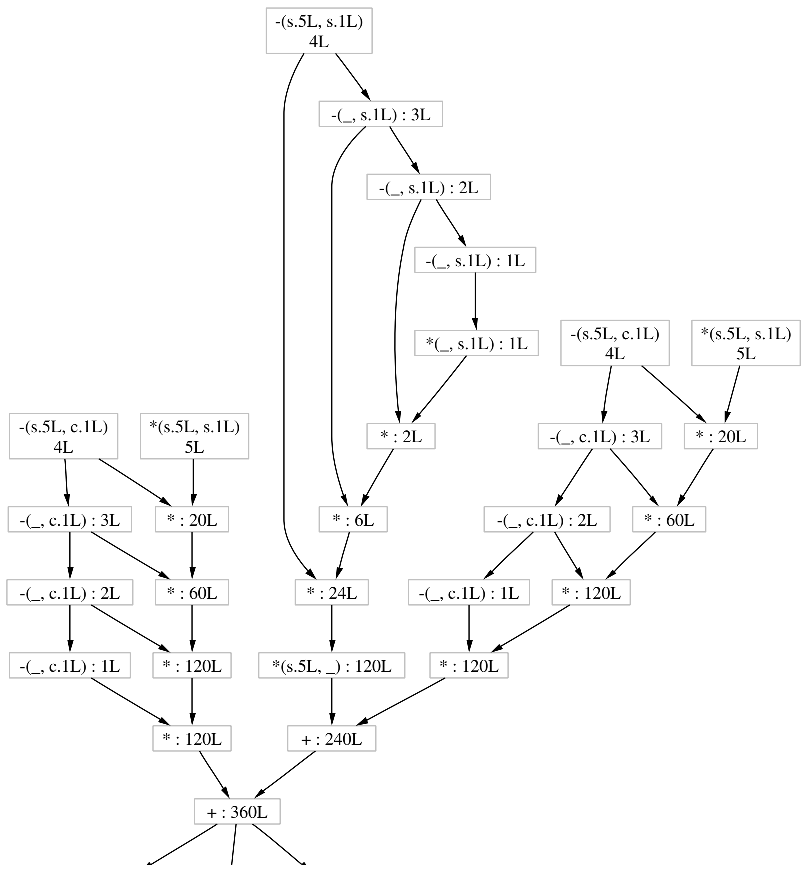

FIGURE 5.7 Dynamic data flow graph for three factorial(5) functions