Candidate for Index : This is not exhaustive and just to help the reflexion on what to index.

Candidate for modification

First appearance of a notion in the text. Should be '\emph'ed

CHAPTER 1

Semantics

L. Beringer

Progress: 90%

Text and figures formating in progress

Introduction is a bit too long

In this chapter we discuss alternative representations that highlight aspects of control and data flow structure that lie at the heart of the SSA discipline. The development of such representations is motivated by the following considerations.

- Firstly, the reformulation of the SSA discipline in other formalisms complements the intuitive meaning of "unimplementable" ϕ-instructions and makes syntactic conditions and semantic invariants that are implicit in the definition of SSA more explicit. The introduction of SSA itself was motivated by a similar goal: to represent aspects of program structure, namely the def-use relationships, explicitly in syntax, by enforcing a particular naming discipline. In a similar way, the functional representations treated in this chapter immediately enforce invariants such as "all ϕ-functions in a block must be of the same arity", "the variables assigned to by the ϕ-functions in a block must be distinct", "ϕ-functions are only allowed to occur at the beginning of a basic block", or "each use of a variable should be dominated by its (unique) definition". Consequently, less code is required that validates the well-structuredness of programs between compiler phases, and the robustness and maintainability of compiler frameworks are increased.

Not actually true, e.g., IR Cliff Click

- Secondly, alternative representations provide formal criteria with respect to which SSA-based code transformations may be proven correct. For example, Glesner [14] uses a representation of SSA in terms of abstract state machines to prove the correctness of a code generation transformation, while Chakravarty et al. [10] prove the correctness of a functional representation of Wegmann and Zadeck's SSA-based sparse conditional constant propagation algorithm [40]. In the same spirit, these representations provide a formal basis for comparing variants of SSA – such as the variants discussed elsewhere in this book –, for translating between these variants, and for constructing and destructing SSA.

The main body of the chapter should contain as few ref as possible, to be more "tutorial"-like. I.e., the main goal is to explain things, not to give credit / be exhaustive. This is like a course and we don't have to justify what we say in the main text.
This parts looks too much like an article.
Credit/Refs should be given in the last section ("further reading") of the chapter.

(this comments applies wherever there are other refs in the chapter)

- Thirdly, they facilitate the implementation of interpreters operating at SSA level. This enables the compiler developer to experimentally validate SSA-based analyses and transformations at their genuine language level, prior to SSA destruction.
- Finally, alternative formalisms provide conceptual insight into for the appeal of SSA by relating its core ideas to concepts from other areas of compiler and programming language research.

We first outline representations in functional programming languages. Pioneered by O'Donnell, Kelsey, and Appel [25, 20, 6], these representations are based on the correspondences between the control flow structure of SSA and a functional programming discipline called continuation-passing-style (short: CPS), and between the constraints on def-use-relationships imposed by ϕ-nodes and the notion of static scope. We examine various aspects of these correspondences in detail, outline how the construction and destruction of SSA can be mirrored by matching operations on functional programs, and indicate how the correspondence may be extended to program analysis frameworks. useless

We then consider a representation of SSA programs as sets of mutually recursive equations $x_i = e_i$ that stresses data flow aspects of SSA and was originally introduced by Pop [29]. Our discussion focuses on some principal underlying this representation, preparing for a more in-depth discussion of advanced topics in Chapter **??**.

??? Don't understand, and don't think it can be understood in an introduction.

We conclude by discussing some pointers to the literature. useless

1.1 Functional interpretations

Our first model of SSA is provided by functional programming languages and exploits the observation that core properties of SSA have direct counterparts in the world of functional programming. We first describe the functional representations *continuation-passing* and *direct style* and then outline the relationhip to SSA by discussing functional counterparts to the construction and destruction of SSA code.

Like the remainder of the book, our discussion concerns code in a single procedure.

1.1.1 Low-level functional program representations

Functional languages represent a procedure by a declaration

$$\texttt{function}\ f(x_0, \ldots, x_n) = e$$

where the syntactic category of expressions e conflates the notions of expressions and commands of imperative languages.

paragraph mode, It's ok, I take care of the style (Florent)

Variable assignment versus name binding

A language construct provided by almost all functional languages is the let-binding \emph

$$\texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end}.$$

The effect of this expression is to evaluate e_1 and bind the resulting value to variable x for the duration of the evaluation of e_2. The code affected by this binding, e_2, is called the *static scope* of x and is easily syntactically identifiable. In the following, we occasionally indicate scopes by code-enclosing boxes, indicating the variables that are in scope using subscripts.

Give example on the 1.1 code

In contrast to an assignment in an imperative language, a let-binding for variable x hides any previous value bound to x for the duration of evaluating e_2 but does not permanently overwrite it. Thus, bindings are treated in a stack-like fashion, and boxes in our code excerpts are properly nested. For example, in code

$$\begin{array}{l} \texttt{let } v = 3 \texttt{ in} \\ \quad \boxed{\begin{array}{l} \texttt{let } y = (\texttt{let } v = 2 * v \texttt{ in } \boxed{4 * v}_{v} \texttt{ end}) \\ \texttt{in } \boxed{y * v}_{v,y} \texttt{ end} \end{array}}_{v} \\ \texttt{end} \end{array} \tag{1.1}$$

the inner binding of v to value $2 * 3 = 6$ shadows the outer binding of v to value 3 precisely for the duration of the evaluation of the expression $4 * v$. Once this evaluation has terminated (resulting in the binding of y to 24), the binding of v to 3 comes back into force, yielding the overall result of 72.

The concepts of binding and static scope ensure that functional programs enjoy the characteristic feature of SSA, namely the fact that each use of a variable is uniquely associated with a point of definition. Indeed, the point of definition for a use of x is given by the *nearest enclosing binding of* x. Occurrences of variables in an expression that are not enclosed by a binding are called *free*. A well-formed procedure declaration contains all free variables of its body amongst its formal parameters. Thus, the notion of scope makes explicit a crucial invariant of SSA that is often left implicit: each use of a variable should be dominated by its (unique) definition.

In contrast to SSA, functional languages achieve the association of definitions to uses without imposing the global uniqueness of variables, as witnessed by the duplicate binding occurrences for v in the above code. As a consequence of this decoupling, functional languages enjoy a strong notion of referential transparency: the choice of x as the variable holding the result of e_1 depends only on the free variables of e_2. In fact, code (1.1) is equivalent to the fragments

+ "we could have used different names in" ... "which is equivalent to..."

$$\texttt{let } v = 3 \texttt{ in } \boxed{\begin{array}{l}\texttt{let } y = (\texttt{let } z = 2 * v \texttt{ in } \boxed{4 * z}_{v,z} \texttt{ end}) \\ \texttt{in } \boxed{y * v}_{v,y} \texttt{ end}\end{array}}_{v} \texttt{ end} \tag{1.2}$$

and

$$\texttt{let } v = 3 \texttt{ in } \boxed{\begin{array}{l}\texttt{let } y = (\texttt{let } y = 2 * v \texttt{ in } \boxed{4 * y}_{v,y} \texttt{ end}) \\ \texttt{in } \boxed{y * v}_{v,y} \texttt{ end}\end{array}}_{v} \texttt{ end} \tag{1.3}$$

These

that are obtained if we rename the bound variable v in a process called *α-renaming*. Code (1.3) illustrates that additional bindings of x in e_1 do not clash with the outer binding of x in `let` $x = e_1$ `in` e_2 `end` as the evaluation of e_1 precedes the binding of its result to x.

It is important? Moreover, the "x, e1, e2" add confusion.

In order to illustrate that the choice of a let-bound variable depends on the free variables of e_2, let us consider possible renamings for the variable y in (1.1). Since the scope of y, namely the expression $y * v$, contains the variable v free, we cannot replace y by v but only by some other variable, say y'.

Seems straightforward and unimportant detail.

Program analyses for functional languages are typically compatible with α-renaming in that they behave equivalently for fragments that differ only in their choice of bound variables, and program transformations rename bound variables whenever necessary.

A consequence of referential transparency, and thus a property typically enjoyed by functional languages, is *compositional equational reasoning*: the meaning of a piece of code is only dependent on the meaning of its subexpressions. Hence, languages with referential transparency allow one to replace a subexpression by some semantically equivalent phrase without altering the meaning of the surrounding code. Since semantic preservation is a core requirement of program transformations, the suitability of SSA for formulating and implementing such transformations can be explained by the proximity of SSA to functional languages.

"i.e., ..." (I am not sure of what that means)

O'Donnell [25] and Kelsey [20] observed that the correspondence between let-bindings and points of variable definition in assignments extends to other aspects of program structure, in particular to code in *continuation-passing-style* (CPS), a program representation routinely used in compilers for functional languages [35, 5].

Control flow: continuations

Satisfying a roughly similar purpose as return addresses or function pointers in imperative languages, a continuation specifies how the execution should proceed once the evaluation of the current code fragment has terminated. Syntactically, continuations are expressions that may occur in functional position (i.e., are typically applied to argument expressions), as is the case for the variable k in

$$\begin{array}{l}\texttt{let}\ v = 3\ \texttt{in}\\ \quad \texttt{let}\ y = (\texttt{let}\ v = 2 * v\ \texttt{in}\ 4 * v\ \texttt{end})\\ \quad \texttt{in}\ k(y * v)\ \texttt{end}\\ \texttt{end}\end{array}\; . \tag{1.4}$$

In effect, k represents any function that may be applied to the result of expression (1.1).

Surrounding code may specify the conrecte continuation by binding k to a suitable expression, as in

concrete

Expl of \lambda terms should come before the use, with the quivalent function def (function k(x) = 2*x)

$$\begin{array}{l}\texttt{let}\ k = \lambda\, x.\, 2 * x\\ \texttt{in}\ \texttt{let}\ v = 3\ \texttt{in}\\ \qquad \texttt{let}\ y = (\texttt{let}\ z = 2 * v\ \texttt{in}\ 4 * z\ \texttt{end})\\ \qquad \texttt{in}\ k(y * v)\ \texttt{end}\\ \quad \texttt{end}\\ \texttt{end}\end{array}$$

or wrap fragment (1.4) in a function definition with formal argument k and construct the continuation in the calling code:

$$\begin{array}{l}\texttt{function}\ f(k) =\\ \quad \texttt{let}\ v = 3\ \texttt{in}\\ \qquad \texttt{let}\ y = (\texttt{let}\ z = 2 * v\ \texttt{in}\ 4 * z\ \texttt{end})\\ \qquad \texttt{in}\ k(y * v)\ \texttt{end}\\ \quad \texttt{end}\\ \texttt{in}\ \texttt{let}\ k = \lambda\, x.\, 2 * x\ \texttt{in}\ f(k)\ \texttt{end}\\ \texttt{end}.\end{array} \tag{1.5}$$

In both cases, the λ-term $\lambda\, x.\, 2 * x$ stipulates that the result of f should be multiplied by 2.

Typically, the caller of f is itself parametric in *its* continuation, as in

$$\begin{array}{l}\texttt{function}\ g(k) =\\ \quad \texttt{let}\ k' = \lambda\, x.\, k(2 * x)\ \texttt{in}\ f(k')\ \texttt{end}.\end{array} \tag{1.6}$$

where f is invoked with a newly constructed continuation k' that applies the multiplication by 2 to its formal argument x (which at runtime will hold the result of f) before passing the resulting value on as an argument to the outer continuation k. In a similar way, the function

```
function h(y, k) =
  let x = 4 in
    let k′ = λz.k(z ∗ x)
    in if y > 0
      then let z = y ∗ 2 in k′(z) end
      else let z = 3 in k′(z) end
    end
  end
```
(1.7)

constructs from k a continuation k' that is invoked (with different arguments) in each branch~~e~~ of the conditional. In effect, the sharing of k' amounts to the definition of a control flow merge point, as indicated by the CFG corresponding to h in Figure 1.1 (left).

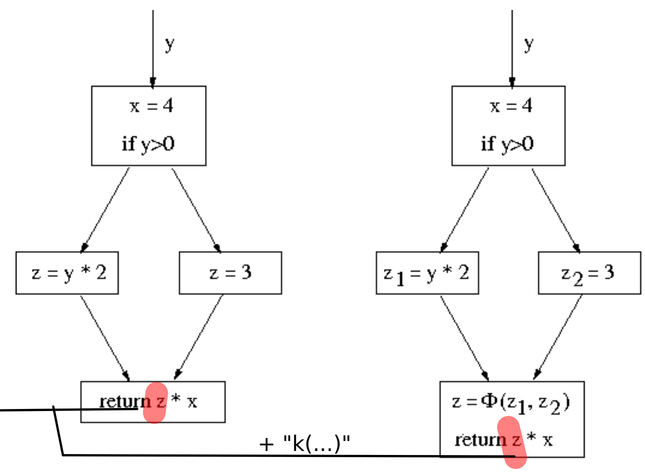


Fig. 1.1 Control flow graph for code (1.7) (left), and SSA representation (right)

The SSA form of this CFG is shown in Figure 1.1 on the right. If we apply similar renamings of z to z_1 and z_2 in the two branches of (1.7), we obtain

```
function h(y, k) =
  let x = 4 in
    let k′ = λz.k(z ∗ x)
    in if y > 0
      then let z₁ = y ∗ 2 in k′(z₁) end
      else let z₂ = 3 in k′(z₂) end
    end
  end
```
(1.8)

We observe that the role of the formal parameter z of continuation k' is exactly that of a ϕ-function: to unify the arguments stemming from various calls sites by binding them to a common name for the duration of the ensuing code fragment – in this case just the return expression. As expected from the above understanding of scope and dominance, the scopes of the bindings for z_1 and z_2 coincide with the dominance regions of the identically named imperative variables: both terminate at the point of function invocation / jump to the control flow merge point.

The fact that transforming (1.7) into (1.8) only involves the referentially transparent process of α-renaming indicates that program (1.7) already contains the essential structural properties that SSA distills from an imperative program.

Programs in CPS equip *all* functions declarations which continuation arguments. By interspersing ordinary code with continuation-forming expressions as shown above, they model the flow of control exclusively by communicating, constructing, and invoking continuations.

Before examining further aspects of the relationship between CPS and SSA, we discuss a close relative of CPS, the so-called *direct-style* representation.

Control flow: direct style

An alternative to the explicit passing of continuation terms via additional function arguments is to represent code as a set of locally named tail-recursive functions. Appel [6] popularized the correspondence to SSA of this representation, which is called *direct style* [32].

In direct style, code (1.7) may be represented as

Why not use the lambda notation anymore? It makes the reader thinks this is where lies the difference

$$\texttt{function}\ h(y) = \boxed{\begin{array}{l} \texttt{let}\ x = 4\ \texttt{in} \\ \boxed{\begin{array}{l}\texttt{function}\ h'(z) = \boxed{z*x}_{x,y,z,h,h'} \\ \texttt{in}\ \boxed{\begin{array}{l}\texttt{if}\ y>0\\ \texttt{then let}\ z = y*2\ \texttt{in}\ \boxed{h'(z)}_{x,y,z,h,h'}\ \texttt{end}\\ \texttt{else let}\ z=3\ \texttt{in}\ \boxed{h'(z)}_{x,y,z,h,h'}\ \texttt{end}\end{array}}_{x,y,h,h'} \\ \texttt{end}\end{array}}_{x,y,h} \\ \texttt{end}\end{array}}_{y,h} \tag{1.9}$$

Maybe add a "in k(h(y))" ?

where the local function h' plays a similar role as the continuation k' and is jointly called from both branches. In contrast to the CPS representation the body of h' returns its result directly rather than by passing it on as an argument to some continuation. Neither the declaration of h nor that of h' contain additional continuation parameters.

The difference is small, and I don't see where it is going

A stricter format is obtained if the granularity of local functions is required to be that of basic blocks:

$$
\begin{array}{l}
\texttt{function}\ h(y) = \\
\quad \texttt{let}\ x = 4\ \texttt{in} \\
\qquad \texttt{function}\ h'(z) = z * x \\
\qquad \texttt{in if}\ y > 0 \\
\qquad\quad \texttt{then function}\ h_1() = \texttt{let}\ z = y * 2\ \texttt{in}\ h'(z)\ \texttt{end} \\
\qquad\qquad\quad \texttt{in}\ h_1()\ \texttt{end} \\
\qquad\quad \texttt{else function}\ h_2() = \texttt{let}\ z = 3\ \texttt{in}\ h'(z)\ \texttt{end} \\
\qquad\qquad\quad \texttt{in}\ h_2()\ \texttt{end} \\
\qquad \texttt{end} \\
\quad \texttt{end}
\end{array}
\tag{1.10}
$$

Now, function invocations correspond precisely to jumps, reflecting more directly the CFG from Figure 1.1. Both, CPS and direct style are compatible with the strict notion of basic blocks as well as more relaxed ones where, for example, functions that have only a single invocation site are inlined (extended basic blocks). At the other expreme, both representation also admit the explicit naming of all control flow points, i.e., the introduction of one local function or continuation per instruction. The questions whether CPS or direct style should be preferred, and what the most appropriate granularity level of functions is, have received considerable attention, with no clear consensus being established. In our discussion below, we employ the arguably easier-to-read direct style, although the gist of the discussion applies equally well to CPS.

Independent of the granularity level of local functions, the process of moving from the CFG to the SSA form is again captured by suitably α-renaming the bindings of z in h_1 and h_2:

$$
\begin{array}{l}
\texttt{function}\ h(y) = \\
\quad \texttt{let}\ x = 4\ \texttt{in} \\
\qquad \texttt{function}\ h'(z) = z * x \\
\qquad \texttt{in if}\ y > 0 \\
\qquad\quad \texttt{then function}\ h_1() = \texttt{let}\ z_1 = y * 2\ \texttt{in}\ h'(z_1)\ \texttt{end} \\
\qquad\qquad\quad \texttt{in}\ h_1()\ \texttt{end} \\
\qquad\quad \texttt{else function}\ h_2() = \texttt{let}\ z_2 = 3\ \texttt{in}\ h'(z_2)\ \texttt{end} \\
\qquad\qquad\quad \texttt{in}\ h_2()\ \texttt{end} \\
\qquad \texttt{end} \\
\quad \texttt{end}
\end{array}
\tag{1.11}
$$

Again, the role of the formal parameter z of the control flow merge point function h' is identical to that of a ϕ-function. In accordance with the fact that the basic blocks representing the arms of the conditional do not contain ϕ-functions, the local functions h_1 and h_2 have empty parameter lists – the free occurrence of y in the body of h_1 is bound at the top level by the formal argument of h.

New paragraph, title "Let-normal-form"?

For both direct style and CPS the correspondence to SSA is most pronounced for code in *let-normal-form*: each intermediate result must be explicitly named by a variable, and function arguments must be names or constants. Syntactically, let-normal-form isolates basic instructions in a separate category of primitive terms a

and then requires let-bindings to be of the form `let` $x = a$ `in` e `end`. In particular, neither jumps (conditional or unconditional) nor let-bindings are primitive. The let-normalized form of (1.2),

Example rendered useless because of below explanation that is easier to understand

$$\begin{array}{l}\texttt{let } v = 3 \texttt{ in}\\ \quad \boxed{\begin{array}{l}\texttt{let } z = 2 * v \texttt{ in}\\ \quad \boxed{\texttt{let } y = 4 * z \texttt{ in } \boxed{y * v}_{v,y,z} \texttt{ end}}_{v,z}\\ \texttt{end}\end{array}}_{v}\\ \texttt{end}\end{array} \tag{1.12}$$

is obtained by pulling the let-binding for z from the e_1-position to the outside of the binding for y. In general, the transformation repeatedly rewrites

$$\begin{array}{l}\texttt{let } x = \texttt{let } y = e_1 \texttt{ in } \boxed{e_2}_{y} \texttt{ end}\\ \texttt{in } \boxed{e_3}_{x}\\ \texttt{end}\end{array} \quad \text{into} \quad \begin{array}{l}\texttt{let } y = e_1\\ \texttt{in } \boxed{\texttt{let } x = e_2 \texttt{ in } \boxed{e_3}_{x,y} \texttt{ end}}_{y}\\ \texttt{end,}\end{array}$$

now become ambiguous because sometimes refer to e1 above, and sometimes to e2 (both in "e1-position") Maybe use "a" position?

subject to the side condition that y not be free in e_3.

Programs in let-normal form thus do not contain let-bindings in the e_1-position of outer let-expressions. The stack discipline in which let-bindings are managed is simplified as scopes are nested inside each other[1]. While still enjoying referential transparency, let-normal code is in closer correspondence to imperative code than nonnormalized code as the chain of nested let-bindings directly reflects the sequence of statements in a basic block, interspersed occasionally by the definition of continuations or local functions.

punctuation before footnote

Summarizing our discussion up to this point, Table 1.1 collects some correspondences between functional and imperative/SSA concepts.

Functional concept	Imperative/SSA concept
variable binding in let	assignment (point of definition)
α-renaming	variable renaming
unique association of binding occurrences to uses	unique association of defs to uses
formal parameter of continuation/local function	ϕ-function (point of definition)
lexical scope of bound variable	dominance region

Table 1.1 Correspondence pairs between functional form and SSA (part I)

[1] Provided the continuation/function definitions are closed, this means that bindings can be implemented destructively, i.e., as assignments/updates.

???

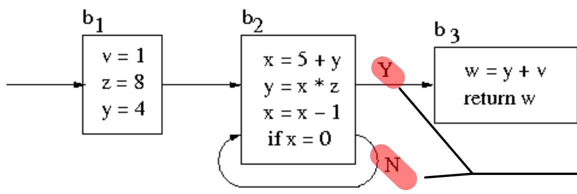
is the remainder of the sentence the definition of "closed"? In either case, I don't understand

1.1.2 *Functional construction of SSA*

The relationship between SSA and functional languages is extended by the correspondences shown in Table 1.2. We discuss some of these aspects by considering the translation into SSA, using the program in Figure 1.2 as a running example.

Functional concept	Imperative/SSA concept
subterm relationship	control flow successor relationship
arity of function f_i	number of ϕ-functions at beginning of b_i
distinctness of formal parameters of f_i	distinctness of LHS-variables in the ϕ-block of b_i
number of call sites of function f_i	arity of ϕ-functions in block b_i
parameter lifting/dropping	addition/removal of ϕ-function
block floating/sinking	reordering according to dominator tree structure
potential nesting structure	dominator tree
nesting level	maximal level index in dominator tree

Table 1.2 Correspondence pairs between functional form and SSA: program structure



true/false for coherence with other chapters

Fig. 1.2 Functional construction of SSA: running example

Initial construction using liveness analysis

A simple way to represent this program in let-normalized direct style is to introduce one function f_i for each basic block b_i. The body of each f_i arises by introducing one let-binding for each assignment and converting jumps into function calls. In order to determine the formal parameters of these functions we perform a liveness analysis. For each basic block b_i, we choose an arbitrary enumeration of its live-in variables. We then use this enumeration as the list of formal parameters in the declaration of the function f_i, and also as the list of actual arguments in calls to f_i. We organize all function definitions in a single block of mutually tail-recursive functions at the top level:

Use line returns so that the same structure as blocks stands out. May Take more space but will not seem obfuscated.

$$\begin{array}{l}
\texttt{function}\ f_1() = \\
\quad \texttt{let}\ v = 1\ \texttt{in let}\ z = 8\ \texttt{in let}\ y = 4\ \texttt{in}\ f_2(v,z,y)\ \texttt{end end end} \\
\texttt{and}\ f_2(v,z,y) = \texttt{let}\ x = 5 + y\ \texttt{in let}\ y = x * z\ \texttt{in let}\ x = x - 1\ \texttt{in} \\
\qquad\qquad \texttt{if}\ x = 0\ \texttt{then}\ f_3(y,v)\ \texttt{else}\ f_2(v,z,y)\ \texttt{end end end} \\
\texttt{and}\ f_3(y,v) = \texttt{let}\ w = y + v\ \texttt{in}\ w\ \texttt{end} \\
\texttt{in}\ f_1()\ \texttt{end}
\end{array} \tag{1.13}$$

The resulting program has the following properties:

- all function declarations are *closed*: the free variables of their bodies are contained in their formal parameter lists[2];
- variable names are not unique, but the unique association of definitions to uses is satisfied;
- each subterm e_2 of a let-binding $\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$ corresponds to the control flow successor of the assignment to x.

If desired, we may α-rename to make names globally unique. As all function declarations are closed, the renamings are independent. The resulting program

+ "(1.14)"

Best not to cut sentences like that when possible.

Same as above. It then can be placed side-by-side with the above example to ease the viewing of diffs.

$$\begin{array}{l}
\texttt{function}\ f_1() = \\
\quad \texttt{let}\ v_1 = 1\ \texttt{in let}\ z_1 = 8\ \texttt{in let}\ y_1 = 4\ \texttt{in}\ f_2(v_1,z_1,y_1)\ \texttt{end end end} \\
\texttt{and}\ f_2(v_2,z_2,y_2) = \texttt{let}\ x_1 = 5 + y_2\ \texttt{in let}\ y_3 = x_1 * z_2\ \texttt{in let}\ x_2 = x_1 - 1\ \texttt{in} \\
\qquad\qquad \texttt{if}\ x_2 = 0\ \texttt{then}\ f_3(y_3,v_2)\ \texttt{else}\ f_2(v_2,z_2,y_3)\ \texttt{end end end} \\
\texttt{and}\ f_3(y_4,v_3) = \texttt{let}\ w_1 = y_4 + v_3\ \texttt{in}\ w_1\ \texttt{end} \\
\texttt{in}\ f_1()\ \texttt{end}
\end{array} \tag{1.14}$$

is an SSA-program in disguise (cf. Figure 1.3): each formal parameter of a function f_i is the target of one ϕ-function for the corresponding block b_i. The arguments of these ϕ-functions are the arguments in the corresponding positions in the calls to f_i. As the number of arguments in each call to f_i coincides with the number of f_i's formal parameters, the ϕ-functions in b_i are all of the same arity, namely the number of call sites to f_i. In order to coordinate the relative positioning of the arguments of the ϕ-functions, we choose an arbitrary enumeration of these call sites.

Under this perspective, the above construction of parameter lists amounts to equipping each b_i with ϕ-functions for all its live-in variables, with subsequent renaming of the variables. Thus, the above method corresponds to the construction of *pruned* (but not minimal) SSA – see Chapter **??**.

em-dash (---)

While resulting in a legal SSA program, the construction clearly introduces more ϕ-functions than necessary. Each superfluous ϕ-function corresponds to the situation where all call sites to some function f_i pass identical arguments. The technique for eliminating such arguments is called λ-*dropping* [12], the inverse of the more widely known transformation λ-*lifting* [17].

ambiguous sentence. Which one is widely known?

[2] Apart from the function identifiers f_i which can always be chosen distinct from the variables

is called

"further reading" section or link to chapters

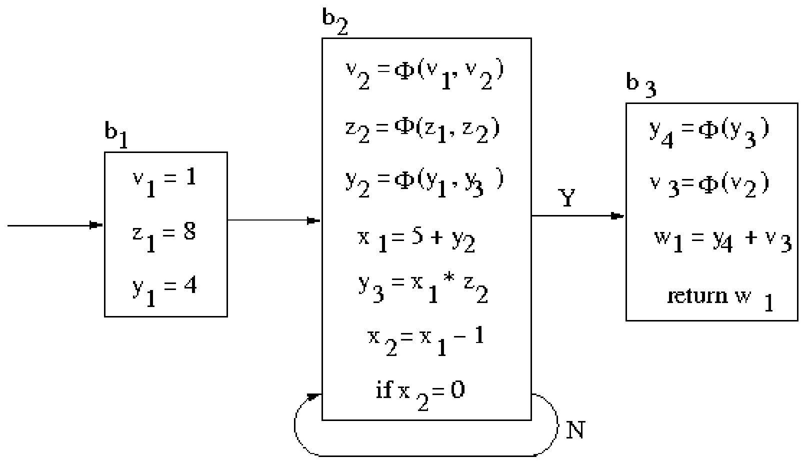


Fig. 1.3 Pruned (non-minimal) SSA form

λ-dropping

λ-dropping may be performed before or after variable names are made distinct, but for our purpose, the former option is more instructive. The transformation consists of two phases, *block sinking* and *parameter dropping*.

Block sinking analyzes the static call structure to identify which function definitions may be moved inside each other. Whenever our set of function declarations contains definitions $f(x_1, \ldots, x_n) = e_f$ and $g(y_1, \ldots, y_m) = e_g$ where $f \neq g$ such that all calls to f occur in e_f or e_g, we can move the declaration for f into that of g. In our example, f_3 is only invoked from within f_2, and f_2 is only called in the bodies of f_2 and f_1. We may thus move the definition of f_3 into that of f_2, and the latter one into f_1.

Several options exist as to where f should be placed in its host function. The first option is to place f at the beginning of g, by rewriting to

$$\texttt{function}\ g(y_1, \ldots, y_m) = \texttt{function}\ f(x_1, \ldots, x_n) = e_f \\ \texttt{in}\ e_g\ \texttt{end}$$

in general and to

Split in two sentences.

$$
\begin{array}{l}
\texttt{function } f_1() = \\
\quad \texttt{function } f_2(v, z, y) = \\
\quad\quad \texttt{function } f_3(y, v) = \texttt{let } w = y + v \texttt{ in } w \texttt{ end} \\
\quad\quad \texttt{in let } x = 5 + y \texttt{ in let } y = x * z \texttt{ in let } x = x - 1 \texttt{ in} \\
\quad\quad\quad \texttt{if } x = 0 \texttt{ then } f_3(y, v) \texttt{ else } f_2(v, z, y) \texttt{ end end end} \\
\quad\quad \texttt{end} \\
\quad \texttt{in let } v = 1 \texttt{ in let } z = 8 \texttt{ in let } y = 4 \texttt{ in } f_2(v, z, y) \texttt{ end end end} \\
\texttt{in } f_1() \texttt{ end}
\end{array}
\tag{1.15}
$$

in the case of our example program. Note that since the declaration of f is closed, moving it into the scope of g's formal parameters $y_1, \ldots, y_m$ (and also into the scope of g itself) is harmless.

A preferable alternative is to insert f near the end of its host function, in the vicinity of its calls:

$$
\begin{array}{l}
\texttt{function } f_1() \ = \\
\quad \texttt{let } v = 1 \texttt{ in let } z = 8 \texttt{ in let } y = 4 \\
\quad \texttt{in function } f_2(v, z, y) = \\
\quad\quad\quad \texttt{let } x = 5 + y \texttt{ in let } y = x * z \texttt{ in let } x = x - 1 \\
\quad\quad\quad \texttt{in if } x = 0 \\
\quad\quad\quad\quad \texttt{then function } f_3(y, v) = \texttt{let } w = y + v \texttt{ in } w \texttt{ end} \\
\quad\quad\quad\quad\quad\quad \texttt{in } f_3(y, v) \texttt{ end} \\
\quad\quad\quad\quad \texttt{else } f_2(v, z, y) \\
\quad\quad\quad \texttt{end end end} \\
\quad\quad \texttt{in } f_2(v, z, y) \texttt{ end} \\
\quad \texttt{end end end} \\
\texttt{in } f_1() \texttt{ end}
\end{array}
\tag{1.16}
$$

This brings the declaration of f additionally into the scope of let-bindings in e_g,

?? def ? not ref transp?

but functional transparency is again respected due to the fact that the declaration is closed.

why param and not \lambda?

The second phase, parameter dropping, removes superfluous parameters based on the syntactic scope structure: a parameter x may be removed from the declaration of and calls to f if

1. the scope in force for x at the declaration of f coincides with the scope in force for x at each call site to f outside f's declaration; and
2. the scope in force for x at any recursive call to f is the one associated with the formal parameter x in f's declaration.

In (1.16), these conditions sanction the removal of both parameters from the non-recursive function f_3. The scope applicable for v at the site of declaration of f_3 and also at its call site is the one rooted at the formal parameter v of f_2. In case of y, the common scope is the one rooted at the let-binding for y in the body of f_2. We thus obtain

$$\begin{array}{l}
\texttt{function } f_1() \; = \\
\quad \texttt{let } v = 1 \texttt{ in let } z = 8 \texttt{ in let } y = 4 \\
\quad \texttt{in function } f_2(v,z,y) = \\
\qquad\quad \texttt{let } x = 5 + y \texttt{ in let } y = x * z \texttt{ in let } x = x - 1 \\
\qquad\quad\; \texttt{in if } x = 0 \\
\qquad\qquad \texttt{then function } f_3() = \texttt{let } w = y + v \texttt{ in } w \texttt{ end} \\
\qquad\qquad\qquad \texttt{in } f_3() \texttt{ end} \\
\qquad\qquad \texttt{else } f_2(v,z,y) \\
\qquad\quad \texttt{end end end} \\
\qquad \texttt{in } f_2(v,z,y) \texttt{ end} \\
\quad \texttt{end end end} \\
\texttt{in } f_1() \texttt{ end}
\end{array} \tag{1.17}$$

Considering the recursive function f_2 next we observe that the recursive call is in the scope of the let-binding for y in f_2's body, preventing us from removing y. In contrast, neither v nor z have binding occurrences in the body of f_2. The scopes applicable at the external call site to f_2 coincide with those applicable at its site of declaration and are given by the scopes rooted in the let-bindings for v and z. Thus, parameters v and z may be removed from f_2, yielding

$$\begin{array}{l}
\texttt{function } f_1() \; = \\
\quad \texttt{let } v = 1 \texttt{ in let } z = 8 \texttt{ in let } y = 4 \\
\quad \texttt{in function } f_2(y) = \\
\qquad\quad \texttt{let } x = 5 + y \texttt{ in let } y = x * z \texttt{ in let } x = x - 1 \\
\qquad\quad\; \texttt{in if } x = 0 \\
\qquad\qquad \texttt{then function } f_3() = \texttt{let } w = y + v \texttt{ in } w \texttt{ end} \\
\qquad\qquad\qquad \texttt{in } f_3() \texttt{ end} \\
\qquad\qquad \texttt{else } f_2(y) \\
\qquad\quad \texttt{end end end} \\
\qquad \texttt{in } f_2(y) \texttt{ end} \\
\quad \texttt{end end end} \\
\texttt{in } f_1() \texttt{ end}
\end{array} \tag{1.18}$$

talked about it before, but not yet defined

as the result of parameter dropping and hence λ-lifting. Interpreting the uniquely-renamed variant of (1.18) back in SSA yields the desired minimal code with a single ϕ-function, for variable y at the beginning of block b_2 – see Figure 1.4. The reason that this ϕ-function can't be eliminated (the fact that y is redefined in the loop) is precisely the reason why y survives λ-dropping.

Is there proof of thi minimality? Ok, we don't want proof here, but it does not seem true if we can construct a program with irreducible CFG. In this case, it is minimal only in the reducible case. Cliff Click uses this method, is it equivalent?

You should only state that, *in this exemple,* it is minimal without going into details, and refs to classical construction chapter.

Given this understanding of parameter dropping we can also see why code (1.16) is preferable to code (1.15): the placement of function declarations in the vicinity of their calls potentially enables the dropping of further parameters, namely those that are let-bound in the host function's body.

An immediate consequence of ϕ-insertion via λ-dropping is that blocks with a single predecessor indeed do not contain ϕ-functions. Such a block b_f is necessarily dominated by its predecessor b_g, hence we can always nest f inside g. Inserting f in

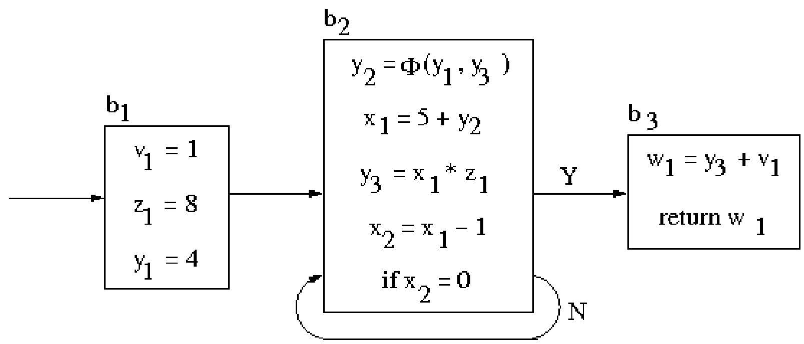


Fig. 1.4 SSA code after λ-dropping

e_g directly prior to its call site implies that condition (1) is necessarily satisfied for all parameters x of f. Thus, all parameters of f can be dropped and no ϕ-function is generated.

Nesting, dominance, loop-closure

Analyzing whether function definitions may be nested inside one another is tantamount to analyzing the imperative dominance structure: function f_i may be moved inside f_j exactly if all non-recursive calls to f_i come from within f_j exactly if all paths from the initial program point to block b_i traverse b_j exactly if b_j dominates b_i. This observation is merely the extension to function identifiers of our earlier remark that lexical scope coincides with the dominance region, and that points of definition/binding occurrences should dominate the uses. Indeed, functional languages do not distinguish between code and data when aspects of binding and use of variables are concerned, as witnesses by our use of the let-binding construct for binding code-representing expressions to the variables k in our syntax for CPS. The fact that they also treat functions like data (as "first class citizens") when argument and result passing are concerned makes functional languages particularly suitable for the extension of (SSA-based) program analyses to inter-procedural analyses[3]. would need some arguments

quotes instead of italics The *optimal* nesting structure is thus given by the dominator tree: the maximal level at which a function may occur is its level (counting from the root) in the dominator tree.

The choice as to where functions are placed corresponds to variants of SSA. For example, loop-closed SSA form **[?, ?]** requires the insertion of ϕ-nodes for all variables that are modified in a loop. This enables one to merge copies of these variables that arise when the loop is unrolled. In the functional setting, this discipline amounts to a small modification of block-sinking: functions f that are called from ref to S. Pop chapter instead

[3] This sentence should probably be moved to the subsection on type systems, once that has been written.

Ref to S. Hack chapter (on update of SSA) within a *recursive* function g are placed at the *same* level as g rather than *inside* g. Returning to our running example, we place f_3 at the same level as f_2, in contrast to code (1.16). The placement of both functions relative to f_1 is left unaltered.

$$
\begin{array}{l}
\texttt{function } f_1() \ = \\
\quad \texttt{let } v = 1 \texttt{ in let } z = 8 \texttt{ in let } y = 4 \\
\quad \texttt{in function } f_3(y,v) = \texttt{let } w = y + v \texttt{ in } w \texttt{ end} \\
\qquad \texttt{and } f_2(v,z,y) = \\
\qquad\quad \texttt{let } x = 5 + y \texttt{ in let } y = x * z \texttt{ in let } x = x - 1 \\
\qquad\quad \texttt{in if } x = 0 \texttt{ then } f_3(y,v) \texttt{ else } f_2(v,z,y) \\
\qquad\quad \texttt{end end end} \\
\qquad \texttt{in } f_2(v,z,y) \texttt{ end} \\
\quad \texttt{end end end} \\
\texttt{in } f_1() \texttt{ end}
\end{array}
\tag{1.19}
$$

As a consequence, any parameter of f that is rebound in g cannot be dropped. In the example, y is not deleted from the parameter list of f_3, as the declaration of f_3 is not any longer in the scope of the binding for y that applies at the call to f_3. In contrast, v may still be dropped from f_3, and v and z may be dropped from f_2:

Add also the modified dominance tree, when the scope changes also does the dom tree. Parallel with \phi placement can be given.

$$
\begin{array}{l}
\texttt{function } f_1() \ = \\
\quad \texttt{let } v = 1 \texttt{ in let } z = 8 \texttt{ in let } y = 4 \\
\quad \texttt{in function } f_3(y) = \texttt{let } w = y + v \texttt{ in } w \texttt{ end} \\
\qquad \texttt{and } f_2(y) = \\
\qquad\quad \texttt{let } x = 5 + y \texttt{ in let } y = x * z \texttt{ in let } x = x - 1 \\
\qquad\quad \texttt{in if } x = 0 \texttt{ then } f_3(y) \texttt{ else } f_2(y) \\
\qquad\quad \texttt{end end end} \\
\qquad \texttt{in } f_2(y) \texttt{ end} \\
\quad \texttt{end end end} \\
\texttt{in } f_1() \texttt{ end}
\end{array}
\tag{1.20}
$$

If we unroll f_2 in this loop-closed form by replacing the recursive call to f_2 by one to its copy f_4 – nesting f_4 inside f_2 allows us to immediately parameter-drop y – we obtain

Keep this "y" for clarity, but mention that it can be dropped.

replace instead by \$f_2'\$ and f_3 by f_1'

```
function f1() =
   let v = 1 in let z = 8 in let y = 4
   in function f3(y) = let w = y + v in w end
      and f2(y) =
         let x = 5 + y in let y = x * z in let x = x - 1
         in if x = 0
            then f3(y)
            else function f4() =
                      let x = 5 + y in let y = x * z in let x = x - 1
                      in if x = 0 then f3(y) else f2(y)
                      end end end
                   in f4() and
         end end end
      in f2(y) end
   end end end
in f1() end
```

(1.21)

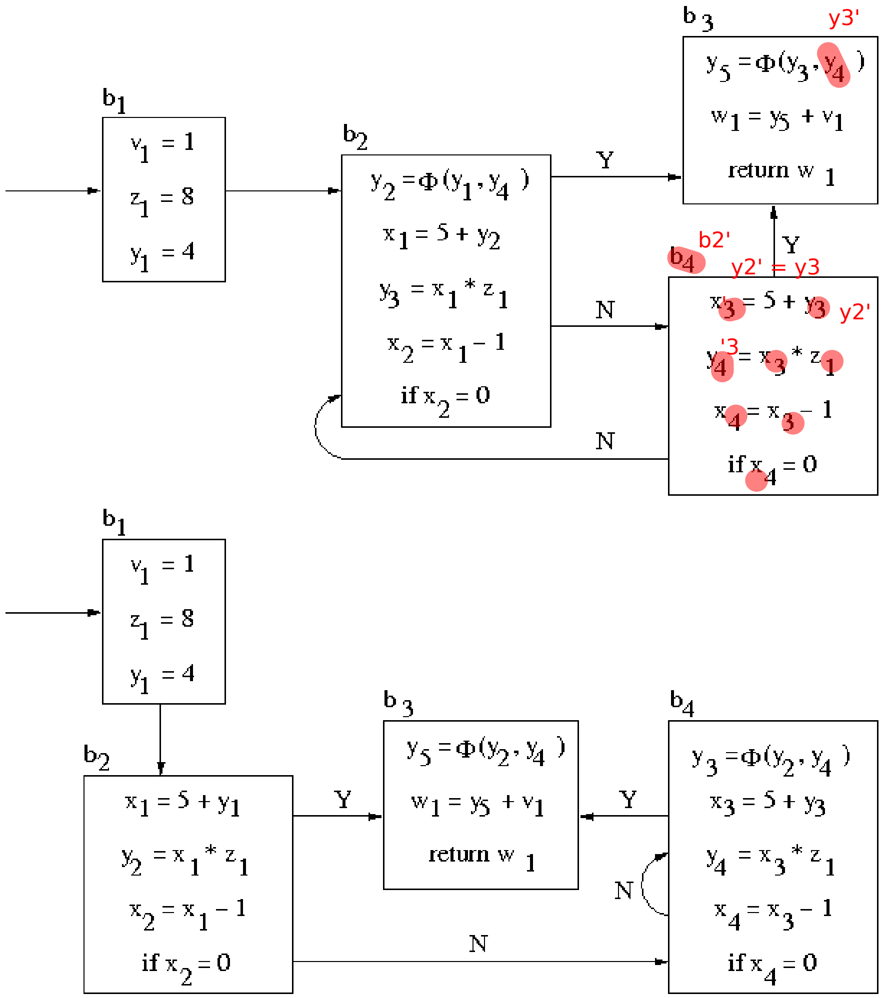
This code exhibits the same sharing as the loop-unrolled SSA program shown at the top of Figure 1.5. The invocations sites to f_3 correspond to the control flow arcs with target b_3, each passing the appropriate value to the “loop closing” parameter y of f_3.

The lower half of Figure 1.5 shows an alternative outcome of loop unrolling. Here, the resulting loop encompasses only b_4. We obtain corresponding functional code if – starting again from (1.20) – we first place the initial copy of f_2 (again named f_4) at the same nesting level as f_2 and replace the call to f_2 inside f_4 by a recursive call to f_4. The only invocation of f_2 that remains is that in f_1, whereas two invocation sites exist for f_4. We then perform λ-dropping, which moves the definition f_4 inside that of f_2 and also drops the parameter y from the declaration of f_2:

see comment on figure 1.5

```
function f1() =
   let v = 1 in let z = 8 in let y = 4
   in function f3(y) = let w = y + v in w end
      and f2() =
         let x = 5 + y in let y = x * z in let x = x - 1
         in if x = 0
            then f3(y)
            else function f4(y) =
                      let x = 5 + y in let y = x * z in let x = x - 1
                      in if x = 0 then f3(y) else f4(y)
                      end end end
                   in f4() and
         end end end
      in f2() end
   end end end
in f1() end
```

(1.22)



etc.

This is not really loop unrolling. I don't see why it is interesting.

Fig. 1.5 Two outcomes of loop-unrolling, corresponding to programs (1.21) and (1.22)

Destruction of SSA

The above example code excerpts where variables are not made distinct exhibit a further pattern: the argument list of any call coincides with the list of formal parameters of the invoked function. This discipline is not enjoyed by functional programs in general, and is often destroyed by optimizaing program transformations. However, programs that do obey this discipline can be immediately converted to imperative non-SSA form. Thus, the task of SSA destruction amounts to converting a functional program with arbitrary argument lists into one where argument lists and formal parameter lists coincide for each function. This is achieved by introducing

let x = v in
let a = z in
let z = y in
let y = a in f(x,y,z)

additional let-bindings of the form $\texttt{let}\ x = y\ \texttt{in}\ e\ \texttt{end}$: for example, a call $f(v, z, y)$ where f is declared as $\texttt{function}\ f(x, y, z) = e$ may be converted to

$$\texttt{let}\ x = y\ \texttt{in let}\ y = z\ \texttt{in let}\ z = x\ \texttt{in let}\ x = a\ \texttt{in}\ f(x, y, z)\ \texttt{end end end end},$$

in correspondence to the move instructions introduced in imperative formulations of SSA destruction (see Chapters **??** and [**?**]). Beringer [8] calls the appropriate transformation on a functional representation GNF-conversion – note that the target language is only syntactically a functional language as it is not immune against α-renaming – and presents a simple local algorithm that considers each call site individually. A single additional variable suffices for performing all necessary insertions of let-bindings, in line with the results of [**?**]. Rideau et al. [34] present an in-depth study of this conversion problem, including a verified implementation in the proof asistant Coq. We don't really care here.

Similar to sequentialization of parallel copies in SSA dest chap

Add pb raised by Sreedhar, copies added before and after (both args & defs renamed)

So first let-bindings in f would be "let x=x' in let y=y' in let z=z'" with f(x',y',z')

1.1.3 *Program analyses*

[4] The correspondence between liveness and free occurrences of names manifests itself in the striking structural similarity between the liveness-equation for assignments

Live

$$\mathsf{LV}([x := a]^i) = \mathsf{Use}(a) \cup (\mathsf{LV}(succ(i)) \setminus \mathsf{Defs}(i))$$

(where i denotes the assignment's program point) and the clause for let-bindings in the definition of free variables,

Free

$$\mathsf{FV}(\texttt{let}\ x = a\ \texttt{in}\ e\ \texttt{end}) = \mathsf{Use}(a) \cup (\mathsf{FV}(e) \setminus \{x\})).$$

In fact, the *least* solution to the liveness equations cannot only be used to determine the formal parameters of the local functions but in fact assigns each program point i (even intermediate ones) exactly the free non-functional variables of the subexpression[5] corresponding to i. ???

[6]

Program analyses for functional languages are typically formulated as *type systems*. Table 1.3 collects some correspondence pairs that relate concepts from type systems to notions from dataflow analysis frameworks. A typical type judgement

Indeed :-)

[4] This section still needs polishing/reformulation

[5] even if variables not globally unique? Add example!

[6] In principle, the parameter lists can be constructed from any solution to the liveness-*in*equations. These arise by replacing = with ⊇ in the dataflow equations. Using inequations rather than equations allows functions to have more formal parameters than strictly necessary. Requiring all parameter lists to be chosen according to the *same* solution prevents (ill-defined) functions whose bodies contain free variables that are not amongst the formal parameters. In particular, including all variables in all parameter lists constitutes a solution to the inequations (and legal functional and SSA programs) but not necessarily a solution to the equations.

Functional concept	Imperative/SSA concept
free variable	live-in variable (least solution)
type systems	dataflow frameworks
typing context	scope-aware symbol table
typing rules	dataflow (in-)equations/transfer functions
subtype relationship	merge operator
typing type inference	fixed point iteration
derivations	solutions to dataflow equations
polymorphism/intersection types	polyvariance

Table 1.3 Correspondence pairs between functional form and SSA: program analyses

definition

$\Gamma \vdash e : \tau$ associates a type τ to an expression e, based on typing assumptions in context Γ. Usually, contexts track the types of (at least) the free names of e, similarly to a symbol table in an imperative analysis. Thus, almost any type system is an extension of the concept of free variables, turning the above relationship between liveness and free variables into an instance of the given analogies. The distinctness of formal parameters, the distinctness of function names in function declaration blocks, and similar syntactic restrictions, may be easily enforced by equipping the corresponding typing rules with additional side-conditions, and are in any case enforced by many functional languages as part of the language definition.

I don't see the link between free names and symtab

A major benefit of SSA for dataflow analyses is the avoidance of variables that have several unrelated uses but happen to be identically named. Even in the absence of globally unique names, this property is enjoyed by type systems, as the adaptation of type contexts in the rule for let-bindings is compatible with referential transparency.

Imperative analysis frameworks employ transfer functions for relating the information associated with adjacent program points. In accordance with the correspondence between the control flow successor relation and the subterm relationship, this role is in type systems played by syntax-directed typing rules. Merge operators at control flow merge points correspond to appropriate notions of subtyping.

The correspondent to fixed point algorithms for obtaining dataflow solutions is type inference. Both tasks proceed algorithmically in a structurally equal fashion, along the control flow successor-/predecessor relationship or sub-/superterm relationship. Finally, *solutions* of dataflow analyses arise when all constraints are met – in type systems, the corresponding notion is that of a successful typing derivation.

As many functional languages support high-order functions, type systems are particularly well suited for formulating inter-procedural analyses[7].

[7] Maybe the author of the chapter on inter-procedural analyses can briefly take up this point, allowing me to insert a forward-reference here?

In the whole section, a lot of redundancy with Pop's chapter.

Since your chapter is already long, may be removed? Can eventually be replaced with 3-4 sentences in the "link" section

1.2 Data flow representation

[8] Our second model of SSA dispenses with the control flow structure entirely, by eliminating any tangible forms of basic blocks or program order. Instead, the model – introduced by Pop [29] – emphasizes dataflow aspects, i.e., the flow of values along the def-use-chains. Programs are represented as collections of defining equations,

$$\begin{aligned} x_1 &= e_1 \\ &\vdots \\ x_n &= e_n \end{aligned}$$

one for each variable x_i. Contrary to the functional representation, the right-hand sides e_i of these equations do not refer to *control flow successors* but to *data flow predecessors*, i.e., to the variables that provide the operands necessary for updating x_i. As a consequence, the order in which equations are presented is irrelevant, and execution proceeds completely data-driven.

In order to transform a sequence of assignments into this form we may apply the approach for converting a basic block into SSA: we introduce a new variable for each assignment, and substitute these variables in the right-hand sides of the instructions according to the data flow. The order of assignments may be permuted arbitrarily, so a sequence like $x := 5;\ y := x + z;\ x := y * 3$ may, for example, be represented by the equations

$$\begin{aligned} x_2 &= y_2 + 3 \\ y_1 &= x_1 + z_1 \\ x_1 &= 5 \end{aligned}$$

(variable z is live-in here).

In order to transform loops, the category of right-hand side expressions e is extended by two novel operations, $\texttt{loop}_\ell(e, e')$ and $\texttt{close}_\ell(e, e')$. Both operations resemble ϕ-instructions, but their arity is independent of any control flow structure: e and e' are expressions and ℓ is an index from $\{1, \ldots, N\}$ where N is the number of while-statements in the original program. An equation

$$x = \texttt{loop}_\ell(e, e')$$

roughly corresponds to the occurrence of a ϕ-node for x at the beginning of a loop in SSA, assigning e to x during the first iteration of loop ℓ and assigning e' to x in later iterations. An equation

$$x = \texttt{close}_\ell(e, e')$$

[8] This section has to be rewritten.

corresponds roughly to a loop-closing ϕ-node for the loop ℓ. Expression e represents the boolean loop condition, and e' represents the value that will be assigned to x when the loop is left, i.e., when e evaluates for the first time to `false`.

For example, the representation of `i = 7; j = 0; while (j < 10) {j = j + i}` (taken from [29]) contains the five equations

$$\begin{aligned} i_1 &= 7 & j_2 &= \mathtt{loop}_1(j_1, j_3) \\ j_1 &= 0 & j_4 &= \mathtt{close}_1(j_2 < 10, j_2) \\ j_3 &= j_2 + i_1 \end{aligned}$$

where 1 is the (trivial) index for the single while-command occurring in the program. In effect, j_4 is assigned the value held in j_2 in that iteration for which $j_2 < 10$ is falsified for the first time, i.e., 14.

In order to formally define concepts such as *for the first time*, the representation is equipped with a semantics that employs so-called *iteration space vectors*: for a program with N loops, such a vector consists of an N-tuple of values, with the value at position ℓ representing the number of iterations of loop ℓ. Given such a vector k, the meaning of a right-hand-side expression e is given recursively as follows:

- constant expressions and arithmetic operators have their standard (iteration space vector independent) meaning.
- a variable x is interpreted by evaluating its defining equation at the iteration space vector k.
- an expression $\mathtt{loop}_\ell(e_1, e_2)$ evaluates to the value of e_1 at k if k at index ℓ is zero. Otherwise, $\mathtt{loop}_\ell(e_1, e_2)$ evaluates to the result of evaluating e_2 at k', where k' is obtained by decrementing index ℓ of k by one.
- an expression $\mathtt{close}_\ell(e_1, e_2)$ evaluates to the value of e_2 at the iteration space vector k' that arises by updating k by the least value x such that e_1 for k' is `false`.

The semantics for the entire program is then given in denotational style, by a mapping that associates each variable to the interpretation of its right-hand side, i.e., to the function that given an iteration space vector evaluates the expression on that vector.

Due to the decrementing of the iteration index in the interpretation of $\mathtt{loop}_\ell(e_1, e_2)$ expressions, an evaluation of this expression for a particular k only requires further evaluations at vectors that are smaller with respect to the component-wise ordering on tuples, with least element $(0, \dots, 0)$. In contrast, the minima mentioned in the interpretation of $\mathtt{close}_\ell(e_1, e_2)$ do not necessarily exist. In each such case, the interpretation of the equation in question, and thus the corresponding variable, is set to $\perp$, in accordance with the standard treatment of non-termination in denotational semantics [41].

In contrast to ϕ-instructions in SSA, the semantics of the equation-based representation thus does not require control-flow information to be maintained, as the choice as to which argument of $\mathtt{loop}_\ell(e, e')$ is evaluated is encoded in the dependency on the entry at the appropriate position in the iteration space vector.

The article [29] and Pop's dissertation [28] contain formal details about the representation, its interpretation, and its formal relationship to a non-SSA language. In particular, these sources explain how a conventional program of assignments and loops may be compositionally converted into a set of equations, in a semantics-preserving way. Source program expressions are uniquely labeled, so that globally unique variable names can be generated by differentiating the original program variables according to the (naturally distinct) labels for assignments. Contrary to the unstructured labels used in [**?**], the authors use a class of labels ("Dewey-like numbers") with the following three properties.

extensibility: this feature is used for generating fresh target variables for ϕ-operations

hierarchical structure according to the subterm relation: this admits a structure-directed translation into the equation-based representation

compatibility with the control flow successor relationship: this feature – in combination with the iteration space vectors – is employed to define a compositional (denotational) semantics of the source language.

In addition to proving a suitable theorem asserting that the translation is semantics-preserving, the authors also give a reading of this result in terms of classical models of computations by interpreting it as the embedding of the RAM model into the model of partial recursive functions. Similar to out-of-SSA translation, a conversion is defined (and proven correct) that transforms systems of equations into imperative programs. Finally, Pop's dissertation [28] describes how a number of program analyses may be phrased in terms of the equational language, including induction variable analysis and other loop optimizations.

1.3 Pointers to the literature

The concept of continuations was introduced multiple times, the earliest discoveries being attributed by Reynolds [33] and Wadsworth [39] to van Wijngaarden [38] and Landin [22]. Early uses and studies of CPS include [31, 27].

The relative merits of the various functional representations remain an active area of research, in particular with respect to their integration with program analyses and optimizing transformations, and conversions between these formats. Recent contributions include [11, 30, 21].

Occasionally, *direct style* refers to the combination of tail-recursive functions and let-normal form. Variations of this discipline include *administrative normal form* (A-normal form, ANF [13]), B-form [36], and SIL [37].

Closely related to continuations and direct-style functional representations are *monadic* languages such as Benton et al.'s MIL [7] and Peyton-Jones et al.'s language [18]. These partition expressions into a category of *values* and *computations*, similar to the isolation of primitive terms in let-normal form (see also [32, 27]).

This allows one to treat side-effects (memory access, IO, exceptions,...) in a uniform way, following Moggi [24].

Regarding formally worked-out instantiations of the correspondences for program analyses, Chakravarty et al. present a functional analysis of sparse constant propagation [10]. Beringer et al. [9] consider data flow equations for liveness and read-only variables, and formally translate their solutions to properties of corresponding typing derivations. Laud et al. [23] present a formal correspondence between dataflow analyses and type systems but consider a simple imperative language rather than SSA or a functional representation. The textbook [**?**] presents a unifying perspective on program analysis techniques, including data flow analysis, abstract interpretation, and type systems.

Modern textbooks on programming language semantics and type systems include [41, 15, 26].

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References

Way too many citations. We are trying to achieve like max 1page or 15 citations. Too many will throw off the reader.

1. volume 28. ACM, June 1993.
2. *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, volume 31. ACM, May 1996.
3. *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 1998. ACM.
4. *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, 1998. SIGPLAN Notices 34(1),January 1999.
5. Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.
6. Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998.
7. Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java Bytecodes. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)* [4], pages 129–140. SIGPLAN Notices 34(1),January 1999.
8. Lennart Beringer. Functional elimination of phi-instructions. *Electronic Notes in Theoretical Computer Science*, 176(3):3–20, 2007.
9. Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. *Electronic Notes in Theoretical Computer Science*, 85(1), 2003.
10. Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. A functional perspective on SSA optimisation algorithms. *Electronic Notes in Theoretical Computer Science*, 82(2), 2003.
11. Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *Journal of Functional Programming*, 17(6):793–812, 2007.
12. Olivier Danvy and Ulrik Pagh Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000.
13. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)* [1], pages 237–247.
14. Sabine Glesner. An asm semantics for ssa intermediate representations. In Zimmermann and Thalheim [42], pages 144–160.
15. Carl Gunter. *Semantics of Programming Languages*. MIT Press, 1992.
16. Ralf Hinze and Norman Ramsey, editors. *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*. ACM, 2007.
17. Thomas Johnsson. Lambda lifting: Treansforming programs to recursive equations. In Jouannaud [19], pages 190–203.
18. Simon Peyton Jones, Mark Shields, John Launchbury, and Andrew Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* [3], pages 49–61.
19. Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture, Proceedings*, volume 201 of *LNCS*. Springer, 1985.

20. Richard Kelsey. A correspondence between continuation passing style and static single assignment form. In *Intermediate Representations Workshop*, pages 13–23, 1995.
21. Andrew Kennedy. Compiling with continuations, continued. In Hinze and Ramsey [16], pages 177–190.
22. Peter Landin. A generalization of jumps and labels. Technical report, UNIVAC Systems Programming Research, August 1965. Reprinted in Higher Order and Symbolic Computation, 11(2):125-143, 1998, with a foreword by Hayo Thielecke.
23. Peeter Laud, Tarmo Uustalu, and Varmo Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292–310, 2006.
24. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
25. Ciaran O'Donnell. *High level compiling for low level machines*. PhD thesis, Ecole Nationale Superieure des Telecommunications, 1994. Available from ftp.enst.fr.
26. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
27. Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
28. Sebastian Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. PhD thesis, Research center for computer science (CRI) of the Ecole des mines de Paris, December 2006. Available from `http://cri.ensmp.fr/people/pop/papers/index.htm`.
29. Sebastian Pop, Pierre Jouvelot, and Georges-Andre Silber. In and out of ssa: a denotational specification. Technical report, July 2007. Available from `http://www.cri.ensmp.fr/classement/doc/E-285.pdf`.
30. John H. Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2-3):161–180, 2002.
31. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717 – 714, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397,1998.
32. John C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 141–156, London, UK, 1974. Springer.
33. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Compututation*, 6(3-4):233–248, 1993.
34. Laurence Rideau, Bernard P. Serpette, and Xavier Leroy. Tilting at windmills with coq: Formal verification of a compilation algorithm for parallel moves. *Journal of Automated Reasoning*, 40(4):307–326, 2008.
35. Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405-439, 1998.
36. David Tarditi, J. Gregory Morrisett, Perry Cheng, Christopher A. Stone, Robert Harper, and Peter Lee. Til: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)* [2], pages 181–192.
37. Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.
38. Adriaan van Wijngaarden. Recursive defnition of syntax and semantics. In T. B. Steel Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13 –24. North-Holland, 1966.
39. Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.
40. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Language Systems*, 13(2):181–210, 1991.
41. Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, February 1993.

42. Wolf Zimmermann and Bernhard Thalheim, editors. *Abstract State Machines 2004. Advances in Theory and Practice, 11th International Workshop (ASM 2004), Proceedings*, volume 3052 of *LNCS*. Springer, 2004.