Chapter 1

Semantics —(*L. Beringer*)

Progress: 53%

Review in progress

In this chapter we discuss models that underpin the SSA discipline and some of the concepts associated with SSA, or embed SSA in alternative program representations. Besides complementing the intuitive meaning of "unimplementable" ϕ-instructions, the models are motivated by the aim

- to make 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, representations of SSA in other formalisms explicitly 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", or "ϕ-functions are only allowed to occur at the beginning of of a basic block". Reasoning using semantic models helps us to understand why such conditions are required and to identify opportunities for relaxing them.
- to have formal criteria with respect to which SSA-based code transformations may be proven correct. For example, Glesner [15] uses a representation of SSA in terms of abstract state machines to prove the correctness of a code generation transformation, while Chakravarty et al. [11] prove the correctness of a functional representation of Wegmann and Zadeck's SSA-based sparse conditional constant propagation algorithm [44].
- to 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, before out-of-SSA-translation is performed
- to provide a formal basis for comparing and integrating variants of SSA (such as the variants discussed elsewhere in this book), for translating between these variants, and for translating into and out of SSA
- to provide conceptual support for the appeal of SSA by relating core concepts to principles well-understood in other domains of compiler and programming language research

which ones?

We categorize the models into two groups. The first group concerns representations that emphasize control-flow aspects of SSA by adhering to the control flow successor relation inside basic blocks or the dominator structure between basic blocks. We discuss aspects of the Appel-Kelsey analogy between SSA and functional representatations in

motivating the content of the chapter: program models / representations / semantics

replace all out of SSA with SSA destruction & into SSA with SSA construction

I do not see the relationship between "CFG aspects" of SSA and functional programming.

1. analogy between SSA & functional prog
2. type based representation

all this is too abstract!

some detail, and summarize a recent type-based representation due to Matsuno and Ohori [26].

The second group of models concerns representations that emphasize data flow over control flow. The first model, due to Glesner, disregards control flow inside basic block but retains it at basic block boundaries, and is phrased in terms of abstract state machines [17]. The second model, due to Pop et al. [33], dispenses with control flow entirely and instead views programs as sets of equations that model the assignment of values to variables in a style reminiscent of partial recursive functions.

We conclude with some brief pointers to additional literature.

3. operationaml semantic with abstract state machines
4. equational representation

1 Control-flow-based representations

1.1 Functional interpretations

Our first model of SSA is provided by functional programming languages. This interpretation rooted in the observation that the central goal of SSA, namely to provide each use of a variable with a unique point of definition, is obtained for free from the concepts of binding and lexical scoping. Like the binding of a variable in λ-calculus, a let-binding `let` $x = e$ `in` e' `end` in a functional language binds the result of e to name x for the duration (scope) e'. Contrary to assignments to imperative variables, such a binding *shadows* earlier bindings of x throughout the evaluation of e', but does not overwrite them[1]. Indeed, the choice of name x is arbitrary and the result of e' is not altered if we capture-avoidingly replace ("α-rename") x by some other fresh name y that does not occur free in e'. As mentioned in Chapter **??**, this notion of *referential transparency* is shared between SSA and functional languages.

let binding + lexical scoping => referential transparency as in SSA

use a different letter for a result and a scope

clash-avoidingly?

O'Donnell [28] and Kelsey [22] observed that the correspondence between name binding and point of variable definition extends to other aspects of program structure. The dominance-based control flow structure of SSA corresponds in a precise way to *continuation-passing-style* (CPS), a program representation routinely used in compilers for functional languages [39,5].

??

I do not see the relationship with the dominance

Programs in CPS explicitly communicate code fragments ("continuation terms") that stipulate where evaluation should continue once the execution of the current fragment has terminated. In this respect, continuations are similar to return addresses in procedure calls. Being particular higher-order functions, continuations may create, apply, and communicate further continuations, enabling the efficient representation of arbitrary control flow. Intra-procedural merge points correspond to invocations of identical (local) continuations, albeit with possibly differing actual arguments. Each formal data parameter of such a continuation plays exactly the same role as a single ϕ-function at the beginning of a code block: to unify the arguments stemming from various calls sites and bind them to a unique name for the duration of the ensuing code fragment.

CPS ~ return addresses in prodedure calls

?

?

An alternative to the use of anonymous continuation terms is to represent a procedure as a set of mutually tail-recursive functions – a representation occasionally referred to as *direct style* [36]. For this variant, which we use below, the correspondence was

tail recursice function => "direct style"

[1] In order to avoid confusion, we refer to identifiers in the functional world as *names*, reserving the term *variable* for the imperative regime.

Use of many technical terms in this section...
I would like an exemple (in imperative language and the correspondance in your model) to illustrate each notion:
- binding & lexical scoping / referential transparency : exemple + english terms
- CPS
- direct style

popularized by Appel [6]. The granularity of functions is roughly that of basic blocks, enabling one to model jumps as function invocations. However, deviations from this discipline are possible, for example by inlining functions that have only a single invocation site, i.e. considering extended basic blocks. The tail-recursiveness of functions implies that no invocation stack needs to be maintained for such local function calls. In contrast, procedure invocations obey the usual frame stack discipline and may be represented as function calls that occur in non-tail position.

a BB => function
jump => tail
function invocation
call => function invocation

tail => no stack

I do not get why let-normal form imposes an order. More precisely why we have no order if not normal.

The correspondence to SSA is most pronounced for direct style or CPS programs that are in *let-normal-form*: each intermediate result must be explicitly named, 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. As a consequence of these restrictions, the conversion of unrestricted code into let-normal form fixes an evaluation order, in a similar way as the linearization of a data- and control flow-dependence graph.

let normal form
=> fixes an order

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 [14]), B-form [40], and SIL [41].

some other flavors...

Translation into and out of SSA The correspondence between SSA and functional languages may be broken down into the correspondence pairs shown in Table 1. We discuss some of these aspects by considering the translation into SSA, using the program in Figure 1 as a running example.

Functional concept	Imperative/SSA concept
name-binding in let	assignment (point of definition)
name-binding in function parameter	assignment by ϕ-function (point of definition)
α-renaming	clash-avoiding variable renaming
unique association of binding occurrences to uses	unique association of defs to uses
lexical scope of name	dominance region of variable
subterm relationship	control flow successor relationship
free name	live-in variable (least solution)
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
conversion to GNF	out-of-SSA translation

Table 1. Correspondence pairs between functional form and SSA: program structure

A simple way to represent this program in a functional language is to introduce one function f_i for each basic block b_i. The body of each f_i arises by introducing one

```
b1 : v := 1      b2 : x := 5 + y      b3 : w := y + v
     z := 8           y = y * z            return w
     y := 4           x := x - 1
     goto b2          if x = 0 b3 b2
```

example

Fig. 1. Correspondence between SSA and functional representation: running example

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 list in calls to f_i. We organize all function definitions in a single block of mutually tail-recursive functions at the top of the functional program:

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

corresponding direct style

The resulting programs has the following properties:

- all function declarations are closed, i.e. the free variables of their bodies are contained in their formal parameter lists
- names are not unique, but the program obeys a core property of SSA: each use of a name is associated with a unique binding, namely the unique *innermost* binding in whose scope the use occurs.
- inside each function the subterm relationship captures the control flow successor relation in the corresponding basic block.

?

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

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

and the clause for the corresponding expression form in the definition of free variables,

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

correspondance btwn liveness & free variables

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

use the same term alpha-rename of table 1 & paragraph 1.1

We may α-convert (i.e. replace a bound name by a fresh one, and substitute the latter for all free occurrences of the former in its scope) this program to make names globally unique. As no function declaration is nested inside another, the renamings are independent from each other. The resulting program is an SSA-program in disguise:

alpha-renaming => SSA

each formal parameter of a function f_i represents the target variable 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. In order to coordinate the relative positioning of the arguments of the ϕ-functions, we choose an arbitrary enumeration of these call sites.

function =~ phi

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[2]. Thus, the above method corresponds to the construction of *pruned* SSA–see Chapter **??**.

pruned non-minimal SSA

chapter properties and flavors that describes pruned SSA (not the construction)

but non minimal

Indeed,

While resulting in a legal SSA program, the construction clearly introduces more ϕ-functions than necessary. A closer inspection reveals that each superfluous ϕ-function corresponds to some function f_i passing one of its arguments on to some other function f_j without modifying it. The technique for eliminating such arguments is called *lambda-dropping* [13] and is the inverse of *lambda-lifting* [19].

reducing #phis: lambda dropping (~ copy folding)

Lambda-dropping may be performed before or after variable names are made distinct, and first analyses the static invocation graph to identify when function definitions may be moved inside each other (*block sinking*). As the above construction yields closed functions, block-sinking is always legal. 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.

block sinking ~= building dom-tree

Several options exist as to where the function definitions should be inserted in their host functions. Inserting them near the beginning yields the program

after block sinking

```
fun f1() =
    fun f2(v, z, y) =
        fun f3(y, v) = let w = y + v in w end
        in let x = 5 + y in let y = x * z in let x = x − 1 in
            if x = 0 then f3(y, v) else f2(v, z, y) end end end
        end
    in let v = 1 in let z = 8 in let y = 4 in f2(v, z, y) end end end
in f1() end
```

[2] 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 but not necessarily one to the equations.

??

An alternative is to insert them near the end of their host functions, directly prior to their use:

```
fun f1() = let v = 1 in let z = 8 in let y = 4 in
           fun f2(v, z, y) = let x = 5 + y in let y = x * z in let x = x − 1 in
                             if x = 0
                             then fun f3(y, v) = let w = y + v in w end
                                  in f3(y, v) end
                             else f2(v, z, y) end end end
           in f2(v, z, y) end end end end
in f1() end
```

The latter placement enables the second phase of lambda-dropping, called *parameter dropping*, to remove variables from the lists of formal parameters based on the syntactic scope. In our example, both parameters of f_3 can be removed; of the three parameters of f_2, only y survives:

toward minimality

not clear how you handled circuits (recursivity). Is it minimal in all cases?

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

Interpreting this program back in SSA yields code that contains a single ϕ-function, for variable y at the beginning of block b_2. The reason that this ϕ-function can't be eliminated (it is redefined in the loop) is precisely the reason why y survives λ-dropping: the innermost scope in force at y's use in the recursive call to f_2 is the one introduced by the binding of y in f_2's body, and is thus different from the scope in force at the point where f_2 is declared.

? (not defined)

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 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. 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, the recently introduced loop-closed SSA form [] requires the insertion of ϕ-nodes for all variables that are modified in a loop, in order to merge copies of these variables that arise when the loop is unrolled. In the functional setting, this amounts to inserting functions f that are called from within a recursive function g at the same level as g. In our example program, f_3 is thus placed at the same level as f_2, but the placement

you can refer to Sebastian Pop's chapter (loops) & James chapter (gated SSA)

of f_2 is left unaltered.

```
fun f1() = let v = 1 in let z = 8 in let y = 4 in
             fun f3(y, v) = let w = y + v in w end
             fun f2(v, z, y) = let x = 5 + y in let y = x * z in let x = x − 1 in
                                 if x = 0 then f3(y, v) else f2(v, z, y) end end end
             in f2(v, z, y) end end end end
in f1() end
```

~ loop closing

As a consequence, any parameter of the invoked function that is rebound in the loop cannot be λ-dropped. In the example, y is not deleted from the parameter list of f_3.

```
fun f1() = let v = 1 in let z = 8 in let y = 4 in
             fun f3(y) = let w = y + v in w end
             fun f2(y) = 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 f2(y) end end end end
in f1() end
```

Subsequently, loop unrolling replaces the recursive call to f_2 by the body of f_2. The resulting code (here in the form where the unrolling is not isolated as a separate function declaration)

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

contains two invocations sites for f_3, in precise correspondence to the control flow arcs in the unrolled SSA program. Each call passes the correct value to the "loop closing" parameter y of f_3.

The above example code excerpts exhibit a further feature: 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 destroyed by the renaming phase that makes names globally distinct. Optimizing program transformations also destroy this discipline, in a similar way as the relationship between SSA variables originating from the same program variable is destroyed by imperative program manipulations–see Chapter **??**. On the other hand, programs satisfying this discipline can be immediately converted to ϕ-free imperative form - all ϕ-functions are trivial. Thus, the task of translating out of SSA amounts to converting a functional program with arbitrary argument lists into one where argument lists and formal parameter lists coincide for each function. Beringer [8] calls this transformation GNF-conversion and

GNF-conversion
=~ SSA destruction

either you are under SSA and you do not follow this discipline or you are not under SSA (no phi in your code) and whatever you do you follow this discipline.

presents a simple local algorithm that considers each call site individually. A single additional name suffices for performing all necessary permutations, in line with the results of [**?**]. Rideau et al. [38] present an in-depth study of this conversion problem, backed-up by a formalization in a theorem prover.

put it last section of chapter ??

Program analyses Program analyses for functional languages are typically formulated as *type systems*. Table 2 collects some correspondence pairs that relate concepts from type systems to notions from dataflow analysis frameworks. A typical type judgement

Functional concept	Imperative/SSA concept
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

Table 2. Correspondence pairs between functional form and SSA: program analyses

$\Gamma \vdash e : \tau$ 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.

type system extension of FV

?

?

not precise enough an example?

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.

?

awkward

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.

merge operator for data-flow =~ subtyping

example

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.

fixed point =~ type inference

As many functional languages support high-order functions, type systems are particularly well suited for formulating inter-procedural analyses[3].

1.2 Type-based representations

you need:
1. to detail more the notations and the concepts
2. illustrate using examples
3. show what it is useful for

I almost understood anything of this paragraph.

var def = type
reaching defs = set of types
context = ?

The above functional representations recast the SSA discipline *syntactically*. An alternative proposed by Matsuno and Ohori [26] is to leave the syntax of programs in non-SSA form and to model the SSA discipline at the level of types. In their analysis, each base type represents the definition point of a variable. Contexts Γ associate program variables with sets of types, modeling the collection of reaching definitions for the variables available at a given program point. Noting that the sets of definitions reaching the use of a variable form a tree where the paths are the control flows from the definitions to the use, the authors admit types to be formulated over type variables whose introduction and use corresponds to the introduction of ϕ-nodes in SSA.

?

????

Formulated for a language of unstructured control flow, the analysis is phrased as a proof theory in sequential sequent calculus style [29]. Judgements take the form $\mathcal{M}, C, \Gamma \vdash B$ where context Γ associates the free program variables of code sequence B (a basic block) with their types, C contains typing specifications for all targets of jumps in B, and $\mathcal{M}$ contains the (possibly recursive) unfolding definitions of all type variables occurring in Γ and C.

Similar to type systems for functional languages, the hypotheses of typing rules concern the immediate successor of the head instruction of the rules' conclusion. The leaves of a typing derivation are formed by axioms that represent return statements or jumps to successor basic blocks.

As each type uniquely identifies a point of definition, typical SSA-related tasks can be represented as algorithms that construct, analyze or modify the type structure of a program, without having to make the program variables themselves syntactically distinct. In particular, as the SSA discipline is only performed on the type level, out-of-SSA-translation is redundant. The authors show the precise correspondence of their type-based representation to programs in SSA form, give a type inference algorithm that corresponds to Das and Ramakrishna's algorithm for SSA construction [**?**], and present type-based analoga to SSA-based dead-code elimination and common subexpression elimination.

2 Dataflow interpretations

The second group of interpretations for SSA is formed by models that emphasize dataflow aspects, i.e. the flow of values along the def-use-chains. Control flow is either disregarded entirely or restricted to the boundaries between basic blocks.

2.1 Glesner's intra-block dataflow model

Glesner [15] presents a model of SSA that retains the control flow structure between basic blocks, and also the phase distinction between the execution of ϕ-instructions and

no chapter on inter-procedural analysis

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

the execution of ordinary instructions in a basic block. The latter, however, proceed data-driven in that their progress is only governed by the availability of operands.

Formally, the model is phrased in terms of *abstract state machines* [17], an algebraic formalism for transition systems that supports the partitioning of states into *static* components and *dynamic* components. Being invariant under execution, the syntax of a program is encoded using the static features and is modeled as a single first-order algebra over a signature of operations, basic blocks, and predicate symbols which encode the basic-block-level CFG. The dynamic aspects of program execution are encoded using the dynamic features: states are modeled as different first-order algebras over a shared signature. Transition rules model the evolution of states by stipulating how – given a fixed static algebra – the dynamic algebras of adjacent states are related.

The dynamic language associates with each (static) instruction a slot for the result value. Additional dynamic components include the representations to the current and adjacent basic blocks, and a tag that distinguishes the ϕ-phase from the phase for non-ϕ- instructions. The transition rules for instructions are predicated on the existence of values in the result slots of the dataflow predecessor instructions, such that instructions that have all argument positions filled may fire in an arbitrary order, updating their result slots. Conditional and unconditional jumps make their result available in slots that are used to update the current-block data structures.

In addition to defining the operational model of SSA, Glesner also defines a similar model for a machine language with unstructured control flow, and then proves the functional correctness of a translation that allocates one register per instruction (holding the result value), eliminates ϕ-instructions by appropriate copy operations, and linearizes the data flow graphs inside basic blocks by topological sorting.

why do we need this formalism?

this is too vague. We need an example to illustrate.

?

?

why is this semantic useful? You need to promote it.

?

program syntax:static comp. execution: dynamic comp.

tokens for CFG

2.2 Pop et al.'s global dataflow model

The model introduced by Pop et al. [33] dispenses with the control flow structure entirely, by eliminating any tangible forms of basic blocks or program order. 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

This is somehow redundant with loop's chapter (from Seb's Pop).

equations

$$x_2 = y_2 + 3$$
$$y_1 = x_1 + z_1$$
$$x_1 = 5$$

(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

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

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 [33]) contains the five equations

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

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.

awkward for what it is actually.

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 $\texttt{loop}_\ell(e_1, e_2)$ evaluates to the value of e_1 at k if k at index ℓ is zero. Otherwise, $\texttt{loop}_\ell(e_1, e_2)$ evaluates to the result of evaluating e_2 at k', where k' is obtained by decrementing index ℓ of k by one.

– an expression $\texttt{close}_\ell(e_1, e_2)$ 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 $\texttt{loop}_\ell(e_1, e_2)$ 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 $(0, \ldots, 0)$. In contrast, the minima mentioned in the interpretation of $\texttt{close}_\ell(e_1, e_2)$ 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 [45].

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 $\texttt{loop}_\ell(e, e')$ is evaluated is encoded in the dependency on the entry at the appropriate position in the iteration space vector.

The article [33] and Pop's dissertation [32] 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 [32] describes how a number of program analyses may be phrased in terms of the equational language, including induction variable analysis and other loop optimizations.

3 Pointers to the literature

The concept of continuations was introduced multiple times, the earliest discoveries being attributed by Reynolds [37] and Wadsworth [43] to van Wijngaarden [42] and Landin [24]. Early uses and studies of CPS include [35,31].

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 [12,34,23].

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 [20]. These partition expressions into a category of *values* and *computations*, similar to the isolation of primitive terms in let-normal form (see also [36,31]). This allows one to treat side-effects (memory access, IO, exceptions,...) in a uniform way, following Moggi [27].

Regarding formally worked-out instantiations of the correspondences for program analyses, Chakravarty et al. present a functional analysis of sparse constant propagation [11]. 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. [25] 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 [45,16,30].

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

References

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. Annalisa Bossi and Michael J. Maher, editors. *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'06)*, 2006.
11. 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.
12. Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *Journal of Functional Programming*, 17(6):793–812, 2007.

13. 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.
14. 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.
15. Sabine Glesner. An asm semantics for ssa intermediate representations. In Zimmermann and Thalheim [46], pages 144–160.
16. Carl Gunter. *Semantics of Programming Languages*. MIT Press, 1992.
17. Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, 1(1):77–111, 2000.
18. 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.
19. Thomas Johnsson. Lambda lifting: Treansforming programs to recursive equations. In Jouannaud [21], pages 190–203.
20. 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.
21. Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture, Proceedings*, volume 201 of *LNCS*. Springer, 1985.
22. Richard Kelsey. A correspondence between continuation passing style and static single assignment form. In *Intermediate Representations Workshop*, pages 13–23, 1995.
23. Andrew Kennedy. Compiling with continuations, continued. In Hinze and Ramsey [18], pages 177–190.
24. 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.
25. 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.
26. Yutaka Matsuno and Atsushi Ohori. A type system equivalent to static single assignment. In Bossi and Maher [10], pages 249–260.
27. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
28. Ciaran O'Donnell. *High level compiling for low level machines*. PhD thesis, Ecole Nationale Superieure des Telecommunications, 1994. Available from ftp.enst.fr.
29. Atsushi Ohori. A proof theory for machine code. *ACM Transactions on Programning Langunage Systems (TOPLAS)*, 29(6), 2007.
30. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
31. Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
32. 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`.
33. 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`.
34. John H. Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2-3):161–180, 2002.

35. 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.
36. 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.
37. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Compututation*, 6(3-4):233–248, 1993.
38. 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.
39. 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.
40. 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.
41. 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.
42. 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.
43. Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.
44. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Language Systems*, 13(2):181–210, 1991.
45. Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, February 1993.
46. 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.