Denotational Semantics for SSA Conversion

Sebastian Pop, Albert Cohen[†], Pierre Jouvelot, Georges-André Silber

CRI, École des mines de Paris, France
[†]INRIA Futurs, France
{pop, jouvelot, silber}@cri.ensmp.fr, Albert.Cohen@inria.fr

Abstract

We present the first formal specification of the SSA form, an intermediate code representation language used in most modern compilers such as GCC or Intel CC, and of its conversion process from imperative languages.

More specifically, we provide (1) a denotational semantics of the SSA, the Static Single Assignment form, (2) a collecting denotational semantics for a Turing-complete imperative language Imp, (3) a non-standard denotational semantics specifying the conversion of Imp to SSA and, most importantly, (4) the proof of the consistency of this transformation, showing that the structure of the memory states manipulated by imperative constructs is preserved in compilers' middle ends that use the SSA form as control-flow data representation. Interestingly, as an unexpected theoretical by-product of our conversion procedure, we offer a new proof of the reducibility of the RAM computing model to the domain of Kleene's partial recursive functions, to which SSA is strongly related.

These fundamental results ensure that the widely used SSA technology is sound. Our formal denotational framework further suggests that the SSA form could become a target of choice for other optimization technologies such as abstract interpretation or partial evaluation. Indeed, since the SSA form is language-independent, the resulting optimizations would be automatically enabled for any source language supported by compilers such as GCC.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational Semantics

General Terms Languages, Theory

Keywords static single assignment, SSA, conversion, RAM model, partial recursive functions theory

1. Introduction

Most modern and widely distributed compilers for imperative and even some functional languages use the SSA form as an intermediate code representation formalism. This Static Single Assignment form [16] is based on a clear separation of control and data information in programs. While the data model is data flow-based, so that no variable is assigned more than once, the control model traditionnally is graph-based, and represents basic blocks linked within a control-flow graph. When more than one path reaches a given block, values may need to be merged; to preserve the functional characteristics of the dataflow model, this is achieved via so-called ϕ-nodes, which assign to a new identifier two possible values, depending on the incoming flow path.

If this formalism is successfully used in both academic (e.g., GCC [8, 17], LLVM [14]) and commercial (Intel CC [11]) compilers, we believe its theoretical foundations are somewhat lacking (see Section 2 for some of the earlier attempts to formally describe such a framework). One of the main goals of our paper is thus to provide what we believe to be a firmer foundation for this ubiquitous intermediate representation format, addressing both the SSA language itself and the conversion process used to translate imperative source code to intermediate SSA constructs.

Our approach is also practical in that we want to address one shortcoming we see in most of the current literature on the SSA form. The original motivation for the introduction of ϕ-nodes was the conditional statements found in imperative programming languages, for which two paths need to be merged when leaving the branches of an alternative. Thus, most of the methods of ϕ-node placement present in the literature omit the related but somewhat different ϕ-nodes that should also logically occur after loops. In practice this is not an issue, since most compilers' middle ends keep control-flow information on the side (e.g., control-flow graphs or continuations) to deal with loop exit conditions.

It is only quite recently, in the GCC community [21], that the need to introduce such additional ϕ-nodes became apparent, in particular when designing algorithms working directly on loop structures. The initial motivation for the use of these extra nodes was mostly practical [20] since they simplified the implementation of some code transformation techniques that required insertion of new edges in the SSA graph structures. This humble beginning may actually even explain why they were overlooked in recent surveys [3], as their role was yet not well understood at the time.

As we shall see in this paper, these "loop-closing ϕ" expressions are in fact crucial to the expressiveness of SSA, providing the key construct that boosts the computational power of the "pure" SSA language, namely a functional dataflow language without additional ad-hoc control-flow information, from primitive recursion to full-fledged partial recursive functions theory. Moreover, the structural nature of the denotational framework we use here in lieu of the traditional graph-based algorithms, in which the distinction between conditional and loop-originating edges is lost, makes this requirement even more compelling.

The structure of the paper is the following. After this introduction, we survey the related work (Section 2). In Section 3, we introduce our Imp programming language, a very basic yet Turing-complete programming language, and provide its standard denotational semantics. In Section 4, we formally present the SSA form, together with its rather straightforward and standard denotational semantics. In these two sections, we use collecting trace-based semantics, which will be required for our later proof. In Section 5, we show how any construct from Imp can be translated to SSA, using a non-standard denotational semantics to specify this conversion process. Section 6 provides our main theorem that shows that Imp and SSA evaluation processes preserve the consistency of memory states. In Section 7, we discuss some consequences of our result, in particular the reduction of RAM programs to partial recursive functions. We look at future work in Section 8 and conclude in Section 9.

2. Related Work

Since the motivation for the introduction of SSA is mostly one built out of experience stemming from the implementation of compilers' middle ends, there is scant work looking at its formal definition and properties. Yet, it is worth mentioning some previous work that offers a couple of different semantics for the SSA form:

- The early papers [5, 6], which introduce the notation for SSA, mostly present informal semantics and proofs for some optimization algorithms based on the SSA representation.
- Kelsey [13] studies the relationship between the SSA form and the functional programming paradigm, providing a somewhat more formal view of the semantic link between these two notions (see also [2]). He defines a non-standard semantics that translates programs in SSA form to continuation-passing style and back to SSA, providing a way to compile functional languages to the SSA, and making it possible to use the SSA optimizing technology on functional languages. In some sense, our work can be viewed as opening a new venue for this approach by formally showing that the imperative programming paradigm can be mapped to the SSA form.
- A similar semantics, based on continuations, is given by Glesner [9]: she gives an abstract state machine semantics for the SSA, and uses an automatic proof checker to validate optimization transformations on SSA. Yet, there is no formal proof provided to ensure the correctness of this mapping between ASM and SSA. We provide a different, denotational, semantics for SSA and use it to prove the correctness of the SSA conversion process for imperative programs.

All the existing definitions of the SSA form in the literature are influenced by the early papers [6] and consider the SSA as a data structure on top of some intermediate representation, e.g., control-flow graphs augmented with a stream of commands belonging to some imperative language, in other words, a decoration on top of some existing compiler infrastructure. In contrast, this paper is the first to give a complete definition of the SSA form, promoting the SSA to the rank of a full-fledged language. An important aspect of the SSA is exposed this way: the SSA is a declarative language, which, as such and contrarily to what its name might imply, has nothing to do with the concept of assignments, a notion only pertinent in imperative languages. This declarative nature explains why it is a language particularly well-suited to specify and implement program optimizations.

3. Imp, an Imperative Programming Language

Since we are interested in this paper by the basic principles underpinning the SSA conversion process, we use a very simple yet Turing-complete imperative language, Imp, based on assignments, sequences and while loops. As is well-known, conditional statements can always be encoded by a sequence of one or two loops, and thus need not be part of our core syntax.

3.1 Syntax

Imp is defined by the following syntax:

$$\begin{array}{rcl} N & \in & Cst \\ I & \in & Ide \\ E & \in & Expr ::= N \mid I \mid E_1 \oplus E_2 \\ S & \in & Stmt ::= I = E \mid S_1 ; S_2 \mid \mathsf{while}_\ell\ E\ \mathsf{do}\ S \end{array}$$

with the usual predefined constants, identifiers and operators $\oplus$. Note that each while loop is decorated with a number label ℓ that uniquely identifies it. This labeling operation is assumed to have been performed at parsing time in a sequential manner[1]; all while loops are thus numbered, say from 1 to m, in a sequential fashion, for identification purposes. We only require that this numbering preserves the sequential textual order of the program. In the rest of the paper, without loss of generality, we assume that the programs under study have a fixed number m of while loops.

Since the SSA semantics encodes recursive definitions of expressions in a functional manner (see Section 4), we found it easier to define the semantics for Imp as a collecting semantics. It gathers for each identifier and program point its value during evaluation. To keep track of such evaluation points, we use both syntactic and iteration space information. Each statement in the program tree is identified by a Dewey-like number, $h \in N^*$; these numbers can be extended as $h.x$, which adds a new dimension to h and sets its value to x. For instance, the top-level statement is 1, while the second statement in a sequence of number h is $h.2$. The statement that directly syntactically follows h is $h+$, and is defined as follows:

$$\begin{array}{rcl} 1+ & = & 2 \\ (h.1)+ & = & h.2 \\ (h.2)+ & = & h+ \end{array}$$

To deal with the distinct iterations of program loops, we use iteration space vectors k: their components represent the values k_ℓ of the m while loop indices at a given execution point (informally, k collects all loop counter values). During evaluation, these vectors are modified, and we note $k[a/\ell]$ the vector obtained from k by replacing the value at index ℓ with a.

To sum up, a program evaluation point p is a pair $(h, k) \in P = N^* \times N^m$ that represents a particular "run-time position" in a program by combining both a syntactic information, h, and a dynamic one, k, for proper localization.[2]

The only requirement on points is that they be lexicographically ordered, with the infix relation $< \in P \times P \rightarrow Bool$ such that $(h_1, k_1) < (h, k) = (k_1 < k \vee (k_1 = k \wedge h_1 < h))$. For any ordered set S, we note $\max_{<x} S$ the maximum element of S that is less than x (or $\perp$ if no such element exists).

3.2 Semantics

As usual, the denotational semantics of Imp operates upon functions f on lattices or CPOs [19]; all the domains we use thus have a $\perp$ minimum element. Following a general convention, we note $f[y/x] = (\lambda a.y \text{ if } a = x, fa \text{ otherwise})$ and $f[z/y/x] = (\lambda a.\lambda b.z \text{ if } a = x \wedge b = y, fab \text{ otherwise})$ the functions that ex-

[1] To conform to our denotational framework, note that this global decoration of the abstract syntax tree can also be specified as a denotational process.

[2] Intuitively, each (h, k) occurs only once in a given execution trace (the ordered sequence of all states).

tends f at a given point x. The domain of f, i.e., the set of values on which it is defined, is given as $Dom\ f = \{x \mid f(x) \neq \perp\}$.

The semantics of expressions uses states $t \in T = Ide \rightarrow P \rightarrow \mathcal{V}$; a state yields for any identifier and evaluation point its numeric value in $\mathcal{V}$, a here unspecified numerical domain for the values. The use of points gives our semantics its collecting status; in some sense, our semantics specifies traces of computation. The semantics $\mathcal{I}[\![]\!] \in Expr \rightarrow P \rightarrow T \rightarrow \mathcal{V}$ expresses that an Imp expression, given a point and a state, denotes a value in $\mathcal{V}$ (we use $in_{\mathcal{V}}$ as the injection function of syntactic constants in $\mathcal{V}$) :

$$\begin{array}{rcl}
\mathcal{I}[\![N]\!]pt & = & in_{\mathcal{V}}(N) \\
\mathcal{I}[\![I]\!]pt & = & R_{<p}(tI) \\
\mathcal{I}[\![E_1 \oplus E_2]\!]pt & = & \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt
\end{array}$$

where the only unusual aspect of this definition is the use of $R_{<x}f = f(\max_{<x} Dom\ f)$, the reaching definition on a given function f. To obtain the current value of a given identifier, one needs to find in the state the last program point prior to the current p at which I has been updated; since we use a collecting semantics, we need to "search" the states for this last definition.

The semantics of statements $\mathcal{I}[\![]\!] \in Stmt \rightarrow P \rightarrow T \rightarrow T$ yields the state obtained after executing the given statement at the given program point, given an incoming state:

$$\begin{array}{rcl}
\mathcal{I}[\![I = E]\!]pt & = & t[\mathcal{I}[\![E]\!]pt/p/I] \\
\mathcal{I}[\![S_1; S_2]\!](h,k) & = & \mathcal{I}[\![S_2]\!](h.2,k) \circ \mathcal{I}[\![S_1]\!](h.1,k)
\end{array}$$

These definitions are rather straightforward extensions of a traditional standard semantics to a collecting case. For an assignment, we add a new binding of Identifier I at Point p to the value of E. A sequence simply composes the transformers associated to S_1 and S_2 at their respective points $h.1$ and $h.2$. And, as usual, we specify the semantics of a while loop as the least fixed point $\text{fix}(W)$ of the W functional defined as:

$$\begin{array}{l}
\mathcal{I}[\![\mathsf{while}_\ell\ E\ \mathsf{do}\ S]\!](h,k) = \text{fix}(W)(h, k[0/\ell]) \\
W = \lambda w.\lambda(h,k).\lambda t. \\
\quad \left\{ \begin{array}{ll} w(h, k_{\ell+})(\mathcal{I}[\![S]\!](h.1,k)t), & \text{if } \mathcal{I}[\![E]\!](h.1,k)t, \\ t, & \text{otherwise.} \end{array} \right.
\end{array}$$

where, as a shorthand, $k_{\ell+}$ is the same as k, except that the value at index ℓ is incremented by one (we use latter $k_{\ell-}$, with a decrement by one).

Beginning with an iteration vector set to 0 for index ℓ, if the value of the guarding expression E is true, we iterate the while loop at an updated point, which uses the same syntactic label as before but an iteration space vector where the value at index ℓ has been incremented, since an additional loop iteration is taking place. If the loop test is false, we simply consider the loop as a no-op.

3.3 Example

To illustrate our results, we will use a single example running throughout this paper; we provide in Figure 1 this simple program written in a concrete syntax of Imp, together with its semantics, i.e., its outgoing state when evaluated from an empty incoming state.

In this example, if we assume that the whole program is at Syntactic Location 1, then the first statement is labelled 1.1 while the rest of the sequence (after the first semi-column) is at 1.2. The whole labelling then proceeds recursively from there. Since there is only one loop, m is 1, and the iteration space vectors have only one component, initialized to (0). Thus, for instance, after two loop iterations, the value of J is 14, and this will cause the loop to terminate. The collecting nature of the semantics is exemplified here by the fact we keep track of all values assigned to each variable throughout the whole computation.

```
I = 7;
J = 0;
while1 J < 10 do
  J = J + I;
```

$$\xrightarrow{\mathcal{I}[\![]\!]1\perp} \quad \begin{array}{l} \mathtt{I} \rightarrow (1.1,(0)) \rightarrow 7 \\ \mathtt{J} \rightarrow \left\{ \begin{array}{l} (1.2.1,(0)) \rightarrow 0 \\ (1.2.2.1,(0)) \rightarrow 7 \\ (1.2.2.1,(1)) \rightarrow 14 \end{array} \right. \end{array}$$

Figure 1. Syntax and semantics for an Imp program.

4. SSA

In the standard SSA terminology [6, 16], an SSA graph is a graph of def-use chains in the SSA form. Each assignment targets a unique variable, and ϕ nodes occur at merge points of the control flow to restore the flow of values from the renamed variables.

Here, we replace this traditional graph-based approach with a programming language-based paradigm; in the SSA form defined below, the ϕ assignments are capturing the characteristics of the control flow, and the usual control-flow primitives consequently become redundant. The use of this self-contained format is one of the new ideas we provide in this paper, which paves the way to a more formal approach to the SSA definition, its conversion process and its correctness.

4.1 Syntax

A program in SSA form is a set of assignments of SSA expressions $E \in SSA$ to SSA identifiers $I_h \in Ide_{\mathsf{SSA}}$. These expressions are defined as follows:

$$\begin{array}{l} E \in SSA ::= \\ \quad N \mid I_h \mid E_1 \oplus E_2 \mid \mathsf{loop}_\ell\phi(E_1, E_2) \mid \mathsf{close}_\ell\phi(E_1, E_2) \end{array}$$

which extend the basic definitions of $Expr$ with two types of ϕ expressions. Note that identifiers I_h in an SSA expression are elements of Ide_{SSA} labeled with a Dewey-like number. Since every assignment in Imp is located at a unique h, this trick ensures that no identifiers in an imperative program will ever appear twice once converted to SSA form, thus enforcing its static single assignment property.

Since we stated that imperative control flow primitives are not part of the SSA representation, we intendedly annotated the ϕ nodes with a label information ℓ that ensures that the SSA syntax is self-contained and expressive enough to be equivalent to any imperative program syntax, as we show in the rest of this paper. ϕ nodes that merge expressions declared at different loop depths are called $\mathsf{loop}_\ell\phi$ nodes and have a recursive semantics. $\mathsf{close}_\ell\phi$ nodes collect the values that come either from the loop ℓ or from before the loop ℓ, when the loop trip count is zero.

More traditional ϕ-nodes, also called "conditional-ϕ" in GCC, are absent from our core SSA syntax since they would only be required to handle conditional statements, which are absent from the syntax of Imp; these nodes would be handled by a proper combination of $\mathsf{loop}\phi$ and $\mathsf{close}\phi$ nodes.

The set of assignments representing an SSA program is denoted in our framework as a finite function $\sigma \in \Sigma = Ide_{\mathsf{SSA}} \rightarrow SSA$ mapping each identifier to its defining expression.

4.2 Semantics

The semantics of an SSA expression $\mathcal{E}[\![]\!] \in SSA \rightarrow \Sigma \rightarrow N^m \rightarrow \mathcal{V}$ provides, given an SSA expression in a program and an iteration space vector, its value. The semantics of an SSA program σ is thus a finite function mapping identifiers I_h to the semantics of their values σI_h.

We give below the denotational semantics of an SSA program tabulated by σ:

$$\begin{array}{rcl}
\mathcal{E}[\![N]\!]\sigma k &=& in_{\mathcal{V}}(N) \\
\mathcal{E}[\![I]\!]\sigma k &=& \mathcal{E}[\![\sigma I]\!]\sigma k \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma k &=& \mathcal{E}[\![E_1]\!]\sigma k \oplus \mathcal{E}[\![E_2]\!]\sigma k \\
\mathcal{E}[\![\mathsf{loop}_\ell\phi(E_1,E_2)]\!]\sigma k &=& \left\{\begin{array}{ll} \mathcal{E}[\![E_1]\!]\sigma k, & \text{if } k_\ell = 0, \\ \mathcal{E}[\![E_2]\!]\sigma k_{\ell -}, & \text{otherwise.} \end{array}\right. \\
\mathcal{E}[\![\mathsf{close}_\ell\phi(E_1,E_2)]\!]\sigma k &=& \mathcal{E}[\![E_2]\!]\sigma \\
&& k[\min\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}/\ell]
\end{array}$$

Constants such as N are denoted by themselves. As can be seen from the definition for identifiers I, we use the traditional syntactic "call-by-text" approach [10] to handle the fixed point nature of an SSA program[3].

$\mathsf{loop}_\ell\phi$ nodes, by their very iterative nature, are designed to represent the successive values of variables successively modified in imperative loop bodies, while $\mathsf{close}_\ell\phi$ nodes compute the final value of such induction variables in loops guarded by test expressions related to E_1.

Of course, when a loop is infinite, there is no iteration that exits the loop, i.e., there is no k such that $\neg\mathcal{E}[\![E_1]\!]\sigma k$, and thus the set $\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}$ is empty. In such a case, $\min\emptyset$ corresponds to $\perp$.

4.3 Example

We informally illustrate in Figure 2 the semantics of SSA using an SSA program intended to be similar to the Imp program provided in Figure 1.

Since by definition SSA uses single assignments, we need to use a different identifier (i.e., subscript) for each assignment to a given identifier (see for instance J) in the Imp program. Of course, all values are functions mapping iteraction vectors to a constant. To merge the two paths reaching in Imp the loop body, we use a $\mathsf{loop}\phi$ expression to combine the initial value of J and its successive iterated values within the loop. A $\mathsf{close}\phi$ expression "closes" the iterative function associated to $\mathtt{J}_2$ to retrieve its final value, obtained when the test expressions evaluates to *false*; in this case, this yields 14.

5. Conversion to SSA

We are now ready to specify how imperative constructs from Imp can be translated to SSA expressions. We use a non-standard denotational framework to specify formally this transformation process.

5.1 Specification

As any denotational specification, our transformation functions use states. These states $\theta = (\mu, \sigma) \in \mathcal{T} = M \times \Sigma$ have two components: $\mu \in M = Ide \to N^* \to Ide_{\mathsf{SSA}}$ maps imperative identifiers to SSA identifiers, yielding their latest SSA names (these can vary since a given identifier I can be used in more than one Imp assignment statement); $\sigma \in \Sigma = Ide_{\mathsf{SSA}} \to SSA$ simply collects the SSA definitions associated to each identifier in the image of M.

The translation semantics $\mathcal{C}[\![]\!] \in Expr \to N^* \to M \to SSA$ for imperative expressions yields the SSA code corresponding to an imperative expression:

[3] Strictly speaking, this denotational semantics is in fact an operational one since it does not use proper structural induction for identifiers (see [19], p.338). A strictly denotational approach could have been defined as a semantical fixed point on a store mapping identifiers to values, but we found that the current formulation leads to a more intuitive wording of our main theorem and proof.

σ:
```
I₁ → 7
J₁ → 0
J₂ → loop₁φ(J₁, J₃)
J₃ → J₂ + I₁
J₄ → close₁φ(J₂ < 10, J₂)
```

$$\Big\downarrow \mathcal{E}[\![]\!]\sigma$$

$$\begin{array}{l}
\mathtt{I}_1 \to \lambda k.7 \\
\mathtt{J}_1 \to \lambda k.0 \\
\mathtt{J}_2 \to \lambda k.\left\{\begin{array}{l} \mathtt{J}_1(k) \text{ for } k = 0 \\ \mathtt{J}_3(k-1) \text{ for } k > 0 \end{array}\right. \\
\mathtt{J}_3 \to \lambda k.\mathtt{J}_2(k) + \mathtt{I}_1(k) \text{ for } k \geq 0 \\
\mathtt{J}_4 \to \lambda k.14
\end{array}$$

Figure 2. Syntax and semantics of ϕ expressions.

$$\begin{array}{rcl}
\mathcal{C}[\![N]\!]h\mu &=& N \\
\mathcal{C}[\![I]\!]h\mu &=& R_{<h}(\mu I) \\
\mathcal{C}[\![E_1 \oplus E_2]\!]h\mu &=& \mathcal{C}[\![E_1]\!]h\mu \oplus \mathcal{C}[\![E_2]\!]h\mu
\end{array}$$

As in the standard semantics for Imp, we need to find the reaching definition of identifiers, although this time, since this is a compile-time translation process, we only look at the *syntactic* order corresponding to Dewey numbers.

The translation semantics of imperative statements $\mathcal{C}[\![]\!] \in Stmt \to N^* \to \mathcal{T} \to \mathcal{T}$ maps conversion states to updated conversion states. The cases for assignments and sequences are straightforward:

$$\begin{array}{rcl}
\mathcal{C}[\![S_1; S_2]\!]h &=& \mathcal{C}[\![S_2]\!]h.2 \circ \mathcal{C}[\![S_1]\!]h.1 \\
\mathcal{C}[\![I = E]\!]h(\mu,\sigma) &=& (\mu[I_h/h/I], \sigma[\mathcal{C}[\![E]\!]h\mu/I_h])
\end{array}$$

since, for sequences, conversion states are simply propagated. For assignments, μ is extended by associating to the imperative identifier I the new SSA name I_h, to which the converted SSA right hand side expression is bound in σ, thus enriching the SSA program with a new binding for I_h.

As expected, most of the work is performed in while loops:

$$\begin{array}{l}
\mathcal{C}[\![\mathsf{while}_\ell\ E\ \mathsf{do}\ S]\!]h(\mu,\sigma) = \theta_2 \text{ with} \\
\quad \theta_0 = (\mu[I_{h.0}/h.0/I]_{I \in Dom\ \mu}, \\
\qquad\quad \sigma[\mathsf{loop}_\ell\phi(R_{<h}(\mu I), \perp)/I_{h.0}]_{I \in Dom\ \mu}), \\
\quad \theta_1 = \mathcal{C}[\![S]\!]h.1\theta_0, \\
\quad \theta_2 = (\mu_1[I_{h.2}/h.2/I]_{I \in Dom\ \mu_1}, \\
\qquad\quad \sigma_1[\mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))/I_{h.0}]_{I \in Dom\ \mu_1} \\
\qquad\qquad [\mathsf{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})/I_{h.2}]_{I \in Dom\ \mu_1})
\end{array}$$

where we note $\theta_i = (\mu_i, \sigma_i)$. We also used the notation $f[y/x]_{x \in S}$ to represent the extension of f to all values x in S with y.

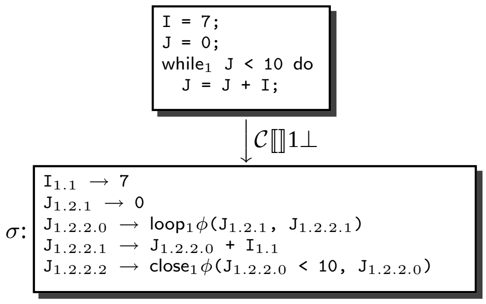
As usual, the conversion process is, by induction, applied on the loop body S located at $h.1$. Yet, this cannot be performed in the original conversion state (μ, σ), since any imperative variable could be further modified in the loop body, creating a new binding which would be visible at the next iteration. To deal with this issue, a new Dewey number is introduced, $h.0$, preceding $h.1$, via which all variables[4] are bound to loopϕ nodes (note that only the SSA expressions corresponding to the control flow coming into the loop can be expressed at that point). It is now appropriate to convert the loop body in this updated conversion state; all references to variables will be to loopϕ nodes, as expected.

Similarly, after the converted loop body, a new Dewey number, $h.2$, following $h.1$, is introduced to bind all variables to closeϕ nodes that represent their values when the loop exits (or $\perp$ if the loop is infinite, as we will see). All references to any identifier once the loop is performed are references to these closeϕ expressions located at $h.2$, which follows, by definition of the lexicographic order on points, all other points present in the loop.

At this time, we are able to provide the entire definition for loopϕ expressions bound at level $h.0$; in particular the proper second subexpression within each loopϕ corresponds to the value of each identifier after one loop iteration.

5.2 Example

We find in Figure 3 the result of the conversion algorithm on our running example; as expected, this SSA program is the same code as the one in Figure 2, up to the renaming of the SSA identifiers. Note that all control-flow information has been removed from the Imp program, thus yielding a “pure”, self-contained SSA form, without any need for additional, on-the-side control-flow data structure.



```
I = 7;
J = 0;
while₁ J < 10 do
  J = J + I;
```

$\downarrow \mathcal{C}[\![]\!]1\perp$

σ:

```
I_1.1   → 7
J_1.2.1 → 0
J_1.2.2.0 → loop₁φ(J_1.2.1, J_1.2.2.1)
J_1.2.2.1 → J_1.2.2.0 + I_1.1
J_1.2.2.2 → close₁φ(J_1.2.2.0 < 10, J_1.2.2.0)
```

Figure 3. Conversion from Imp to SSA.

6. SSA Conversion Consistency

We are finally equipped with all the material required to express our main theorem. Our goal is to prove that our conversion process maintains the memory states consistency between the imperative and SSA representations. This relationship is expressed in the following definition:

DEFINITION 1 (Consistency). *A conversion state $\theta = (\mu, \sigma)$ is* consistent *with the memory state t at point $p = (h, k)$, noted $\mathcal{P}(\theta, t, p)$, iff*

$$\forall I \in Dom\ t, \mathcal{I}[\![I]\!]pt = \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!]\sigma k$$

which specifies that, for any identifier, its value at a given point in the standard semantics is the same as its value in the SSA semantics when applied to its translated SSA equivalent (see Figure 4).

This consistency requirement on identifiers can be straightforwardly extended to arbitrary expressions:

[4] In fact, only the variables modified in the loop body need to be managed this way. We do not worry about such optimization here.

$$\begin{tikzcd} Expr \arrow[r, "\mathcal{C}[\![]\!]h\mu"] \arrow[d, "\mathcal{I}[\![]\!](h,k)t"'] & SSA \arrow[d, "\mathcal{E}[\![]\!]\sigma k"] \\ v \in \mathcal{V} \arrow[r, phantom, "="] & v \in \mathcal{V} \end{tikzcd}$$

Figure 4. Consistency property $\mathcal{P}((\mu, \sigma), t, (h, k))$.

LEMMA 1 (Consistency of Expression Conversion). *Given that $\mathcal{P}(\theta, t, p)$ and an expression $E \in Expr$,*

$$\mathcal{I}[\![E]\!]pt = \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!]\sigma k$$

This directly leads to our main theorem, which ensures the semantic correctness of the conversion process from imperative constructs to SSA expressions (as a shorthand, we note $p+ = (h, k)+ = (h+, k)$):

THEOREM 1 (Consistency of Statement Conversion). *Given any statement S and for all θ, t, p that verify $\mathcal{P}(\theta, t, p)$, if $\theta' = \mathcal{C}[\![S]\!]h\theta$ and $t' = \mathcal{I}[\![S]\!]pt$, the property $\mathcal{P}(\theta', t', p+)$ holds.*

This theorem basically states that if the consistency property is satisfied for any point before a statement, then it is also verified for the statement that syntactically follows it.

PROOF. By induction on the structure of $Stmt$, assuming $\mathcal{P}(\theta, t, p)$:

- for the assignment $[\![I = E]\!]$:

$$\begin{aligned} (\mu', \sigma') &= \mathcal{C}[\![I = E]\!]h\theta = (\mu[I_h/h/I], \sigma[\mathcal{C}[\![E]\!]h\mu/I_h]), \\ t' &= \mathcal{I}[\![I = E]\!]pt = t[\mathcal{I}[\![E]\!]pt/p/I] \end{aligned}$$

$$\begin{aligned} \mathcal{I}[\![I]\!]p{+}t' &= R_{<p+}(t'I) && (\text{def } \mathcal{I}[\![]\!]) \\ &= \mathcal{I}[\![E]\!]pt && (\text{def } t') \\ &= \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!]\sigma k && (\text{Lemma 1}) \\ &= \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!]\sigma' k && (\text{extension of } \sigma') \\ &= \mathcal{E}[\![\sigma' I_h]\!]\sigma' k && (\text{def } \sigma') \\ &= \mathcal{E}[\![I_h]\!]\sigma' k && (\text{def } \mathcal{E}[\![]\!]) \\ &= \mathcal{E}[\![\mu' I h]\!]\sigma' k && (\text{def } \mu') \\ &= \mathcal{E}[\![R_{<h+}(\mu' I)]\!]\sigma' k && (\text{def } R_<) \\ &= \mathcal{E}[\![\mathcal{C}[\![I]\!]h{+}\mu']\!]\sigma' k && (\text{def } \mathcal{C}[\![]\!]) \end{aligned}$$

  The extension to σ' is possible because it does not modify the reaching definitions: $R_{<p}$. So the property holds for I, but it also trivially holds for any $I' \neq I, I' \in Dom\ t$. So, $\mathcal{P}(\theta', t', p+)$ holds.

- for the sequence $[\![S_1; S_2]\!]$:

  Since there are no new bindings between h and $h.1$, $R_{<p} = R_{<(h.1,k)}$ and thus $\mathcal{P}(\theta, t, (h.1, k))$ holds.

  By induction, using the result of the theorem on S_1, with $\theta_1 = \mathcal{C}[\![S_1]\!]h.1\theta$, and $t_1 = \mathcal{I}[\![S_1]\!](h.1, k)t$, the property $\mathcal{P}(\theta_1, t_1, (h.1{+}, k))$ holds.

  Since $h.1+ = h.2$, by induction, using the result of the theorem on S_2, with $\theta_2 = \mathcal{C}[\![S_2]\!]h.2\theta_1$, and $t_2 = \mathcal{I}[\![S_2]\!](h.2, k)t_1$, the property $\mathcal{P}(\theta_2, t_2, (h.2{+}, k))$ holds.

So, the property $\mathcal{P}(\theta', t', p+)$ holds, since $\theta' = \theta_2, t' = t_2$ and $h.2+ = h+$.

- for the loop $\llbracket \mathsf{while}_\ell\ E\ \mathsf{do}\ S \rrbracket$:

  The recursive semantics for while loops suggests to use fixpoint induction ([19], p.213), but this would require us to define new properties and functionals operating on (θ, t, p) as a whole while changing the definition of P to handle ordinals. We prefer to keep a simpler profile here, and give a somewhat ad-hoc but more intuitive proof.

  We will need a couple of lemmas to help us build the proof. As a shorthand, we note $\theta_{ij} = (\mu_i, \sigma_j)$.

LEMMA 2. *With $t = t_0$, $\mathcal{P}_0 = \mathcal{P}(\theta_{12}, t_0, (h.1, k[0/\ell]))$ holds.*

This lemma states that if $\mathcal{P}$ is true at loop entry, then it remains true just before the loop body of the first iteration, at point $(h.1, k[0/\ell])$.

PROOF. $\forall I \in \mathit{Dom}\ t$:

$$\begin{aligned}
&\mathcal{I}\llbracket I \rrbracket(h.1, k[0/\ell])t = \\
&= R_{<(h.1,k[0/\ell])}(tI) && \text{(def } \mathcal{I}\llbracket\rrbracket)\\
&= R_{<p}(tI) && \text{(def } t)\\
&= \mathcal{I}\llbracket I \rrbracket pt && \text{(def } \mathcal{I}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h\mu \rrbracket \sigma k && (\mathcal{P}(\theta, t, p))\\
&= \mathcal{E}\llbracket R_{<h}(\mu I) \rrbracket \sigma k && \text{(def } \mathcal{C}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket R_{<h}(\mu I) \rrbracket \sigma k[0/\ell] && \text{(first iteration)}\\
&= \mathcal{E}\llbracket R_{<h}(\mu I) \rrbracket \sigma_0 k[0/\ell] && \text{(extension to } \sigma_0)\\
&= \mathcal{E}\llbracket \mathsf{loop}_\ell\phi(R_{<h}(\mu I), \bot) \rrbracket \sigma_0 k[0/\ell] && \text{(def } \mathsf{loop}_\ell\phi)\\
&= \mathcal{E}\llbracket \sigma_0 I_{h.0} \rrbracket \sigma_0 k[0/\ell] && \text{(def } \sigma_0)\\
&= \mathcal{E}\llbracket I_{h.0} \rrbracket \sigma_0 k[0/\ell] && \text{(def } \mathcal{E}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mu_0 I h.0 \rrbracket \sigma_0 k[0/\ell] && \text{(def } \mu_0)\\
&= \mathcal{E}\llbracket R_{<h.1}(\mu_0 I) \rrbracket \sigma_0 k[0/\ell] && \text{(def } R_<)\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h.1\mu_0 \rrbracket \sigma_0 k[0/\ell] && \text{(def } \mathcal{C}\llbracket\rrbracket)
\end{aligned}$$

So, $\mathcal{P}(\theta_0, t, h.1, k[0/\ell])$ holds. The extension of θ_0 to θ_{12} concludes the proof of Lemma 2. □

LEMMA 3. *Let $t_x = \mathcal{I}\llbracket S \rrbracket(h.1, k[x-1/\ell])t_{x-1}$. Given $\mathcal{P}_{x-1} = \mathcal{P}(\theta_{12}, t_{x-1}, (h.1, k[x-1/\ell]))$ for some $x \geq 1$, then $\mathcal{P}_x = \mathcal{P}(\theta_{12}, t_x, (h.1, k[x/\ell]))$ holds.*

This second lemma ensures that if $\mathcal{P}$ is true at iteration $x-1$, then it stays the same at iteration x, after evaluating the loop body. Note that the issue of whether we will indeed enter the loop again or exit it altogether is no factor here.

PROOF. By induction, applying the theorem to S, we know that the property

$$\mathcal{P}'_{x-1} = \mathcal{P}(\theta_{12}, t_x, (h.2, k[x-1/\ell]))$$

holds, since $h.1+ = h.2$, and $\theta_{12} = \mathcal{C}\llbracket S \rrbracket h.1\theta_{12}$, as $\mathcal{C}\llbracket\rrbracket$ is idempotent. We thus only need now to "go around" to the top of the loop:

$$\begin{aligned}
&\mathcal{I}\llbracket I \rrbracket(h.1, k[x/\ell])t_x = \\
&= R_{<(h.1,k[x/\ell])}(t_x I) && \text{(def } \mathcal{I}\llbracket\rrbracket)\\
&= R_{<(h.2,k[x-1/\ell])}(t_x I) && \text{(def } R_<)\\
&= \mathcal{I}\llbracket I \rrbracket(h.2, k[x-1/\ell])t_x && \text{(def } \mathcal{I}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h.2\mu_1 \rrbracket \sigma_2 k[x-1/\ell] && (\mathcal{P}'_{x-1})\\
&= \mathcal{E}\llbracket R_{<h.2}(\mu_1 I) \rrbracket \sigma_2 k[x-1/\ell] && \text{(def } \mathcal{C}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I)) \rrbracket \sigma_2 k[x/\ell] && (\mathsf{loop}_\ell\phi)\\
&= \mathcal{E}\llbracket \sigma_2 I_{h.0} \rrbracket \sigma_2 k[x/\ell] && \text{(def } \sigma_2)\\
&= \mathcal{E}\llbracket I_{h.0} \rrbracket \sigma_2 k[x/\ell] && \text{(def } \mathcal{E}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mu_1 I h.0 \rrbracket \sigma_2 k[x/\ell] && \text{(def } \mu_1)\\
&= \mathcal{E}\llbracket R_{<h.1}(\mu_1 I) \rrbracket \sigma_2 k[x/\ell] && \text{(def } R_<)\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h.1\mu_1 \rrbracket \sigma_2 k[x/\ell] && \text{(def } \mathcal{C}\llbracket\rrbracket)
\end{aligned}$$

This concludes the proof of Lemma 3. □

We are now ready to tackle the different cases that can occur during evaluation. These three cases are:

1. when the loop is not executed, that is when the exit condition is false before entering the loop body: we know that $\neg\mathcal{I}\llbracket E \rrbracket t(h.1, k[0/\ell])$. Based on Lemma 2, we can show that $\mathcal{P}(\theta', t', p+)$ holds, as $\theta' = \theta_2$ that extends θ_{12}, $t' = t$ as defined by the exit of the while_ℓ in $\mathcal{I}\llbracket\rrbracket$, and $k[0/\ell] = k$:

$$\begin{aligned}
&\mathcal{I}\llbracket I \rrbracket(p+)t = \\
&= R_{<p+}(tI) && \text{(def } \mathcal{I}\llbracket\rrbracket)\\
&= R_{<p}(tI) && \text{(def } t)\\
&= \mathcal{I}\llbracket I \rrbracket pt && \text{(def } \mathcal{I}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h\mu \rrbracket \sigma k && (\mathcal{P})\\
&= \mathcal{E}\llbracket R_{<h}(\mu I) \rrbracket \sigma k && \text{(def } \mathcal{C}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket R_{<h}(\mu I) \rrbracket \sigma_2 k && \text{(ext. } \sigma_2)\\
&= \mathcal{E}\llbracket \mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I)) \rrbracket \sigma_2 k && \text{(def } \mathsf{loop}_\ell\phi)\\
&= \mathcal{E}\llbracket \sigma_2 I_{h.0} \rrbracket \sigma_2 k && \text{(def } \sigma_2)\\
&= \mathcal{E}\llbracket I_{h.0} \rrbracket \sigma_2 k && \text{(def } \mathcal{E}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mathsf{close}_\ell\phi(\mathcal{C}\llbracket E \rrbracket h.1\mu_1, I_{h.0}) \rrbracket \sigma_2 k && (\mathsf{close}_\ell\phi)\\
&= \mathcal{E}\llbracket \sigma_2 I_{h.2} \rrbracket \sigma_2 k && \text{(def } \sigma_2)\\
&= \mathcal{E}\llbracket I_{h.2} \rrbracket \sigma_2 k && \text{(def } \mathcal{E}\llbracket\rrbracket)\\
&= \mathcal{E}\llbracket \mu_2 I h.2 \rrbracket \sigma_2 k && \text{(def } \mu_2)\\
&= \mathcal{E}\llbracket R_{<h+}(\mu_2 I) \rrbracket \sigma_2 k && \text{(def } R_<)\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I \rrbracket h{+}\mu_2 \rrbracket \sigma_2 k && \text{(def } \mathcal{C}\llbracket\rrbracket)
\end{aligned}$$

2. when the loop is executed a finite number of times, that is when the loop body is executed at least once: let $\omega > 0$ be the first iteration on which the loop condition becomes false:

$$\begin{aligned}
\omega &= \min\{x \mid \neg\mathcal{I}\llbracket E \rrbracket(h.1, k[x/\ell])t_\omega\}\\
&= \min\{x \mid \neg\mathcal{E}\llbracket \mathcal{C}\llbracket E \rrbracket h.1\mu_1 \rrbracket \sigma_2 k[x/\ell]\} && \text{(Lemma 1)}
\end{aligned}$$

   By Lemmas 2 and 3, using induction on S, we know that $\mathcal{P}'_\omega = \mathcal{P}(\theta_{12}, t_\omega, (h.2, k[\omega-1/\ell]))$ holds. We prove below that $\mathcal{P}(\theta', t_\omega, p+)$ also holds (as a shorthand, we note

$k^n = k[n/\ell]$):

$$
\begin{aligned}
&\mathcal{I}\llbracket I\rrbracket(p+)t_\omega = \\
&= R_{<p+}(t_\omega I) && \text{(def } \mathcal{I}\llbracket\rrbracket\text{)}\\
&= R_{<(h.2,k^{\omega-1})}(t_\omega I) && \text{(def } R_<\text{)}\\
&= \mathcal{I}\llbracket I\rrbracket(h.2,k^{\omega-1})t_\omega && \text{(def } \mathcal{I}\llbracket\rrbracket\text{)}\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I\rrbracket h.2\mu_1\rrbracket\sigma_2 k^{\omega-1} && (\mathcal{P}'_\omega)\\
&= \mathcal{E}\llbracket R_{<h.2}(\mu_1 I)\rrbracket\sigma_2 k^{\omega-1} && \text{(def } \mathcal{C}\llbracket\rrbracket\text{)}\\
&= \mathcal{E}\llbracket \mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))\rrbracket\sigma_2 k^{\omega} && (\mathsf{loop}_\ell\phi)\\
&= \mathcal{E}\llbracket \sigma_2 I_{h.0}\rrbracket\sigma_2 k^{\omega} && \text{(def } \sigma_2\text{)}\\
&= \mathcal{E}\llbracket I_{h.0}\rrbracket\sigma_2 k^{\omega} && \text{(def } \mathcal{E}\llbracket\rrbracket\text{)}\\
&= \mathcal{E}\llbracket \mathsf{close}_\ell\phi(\mathcal{C}\llbracket E\rrbracket h.1\mu_1, I_{h.0})\rrbracket\sigma_2 k && (\mathsf{close}_\ell\phi)\\
&= \mathcal{E}\llbracket \sigma_2 I_{h.2}\rrbracket\sigma_2 k && \text{(def } \sigma_2\text{)}\\
&= \mathcal{E}\llbracket I_{h.2}\rrbracket\sigma_2 k && \text{(def } \mathcal{E}\llbracket\rrbracket\text{)}\\
&= \mathcal{E}\llbracket \mu_2 I h.2\rrbracket\sigma_2 k && \text{(def } \mu_2\text{)}\\
&= \mathcal{E}\llbracket R_{<h+}(\mu_2 I)\rrbracket\sigma_2 k && \text{(def } R_<\text{)}\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I\rrbracket h{+}\mu_2\rrbracket\sigma_2 k && \text{(def } \mathcal{C}\llbracket\rrbracket\text{)}
\end{aligned}
$$

Finally, using Kleene's Fixed Point Theorem [19], we can relate the least fixed point fix(W) used to define the standard semantics of while loops and the successive iterations $W^i(\bot)$ of the loop body:

$$
\begin{aligned}
t' &= \text{fix}(W)(h,k[0/\ell])t\\
&= \lim_{i\to\infty} W^i(\bot)(h,k[0/\ell])t\\
&= W^\omega(\bot)(h,k[0/\ell])t\\
&= t_\omega
\end{aligned}
$$

and so $\mathcal{P}(\theta', t', p+)$ holds.

3. when the loop is infinite: $t' = \lim_{i\to\infty} W^i(\bot)(h,k[0/\ell])t = \bot$. Thus:

$$
\begin{aligned}
&\mathcal{I}\llbracket I\rrbracket(p+)\bot = \bot = && \text{(def } \mathcal{I}\llbracket\rrbracket\text{)}\\
&= \mathcal{E}\llbracket \mathsf{close}_\ell\phi(\mathcal{C}\llbracket E\rrbracket h.1\mu_1, I_{h.0})\rrbracket\sigma_2 k && (\min\emptyset = \bot)\\
&= \mathcal{E}\llbracket \sigma_2 I_{h.2}\rrbracket\sigma_2 k && \text{(def } \sigma_2\text{)}\\
&= \mathcal{E}\llbracket I_{h.2}\rrbracket\sigma_2 k && \text{(def } \mathcal{E}\llbracket\rrbracket\text{)}\\
&= \mathcal{E}\llbracket \mu_2 I h.2\rrbracket\sigma_2 k && \text{(def } \mu_2\text{)}\\
&= \mathcal{E}\llbracket R_{<h+}(\mu_2 I)\rrbracket\sigma_2 k && \text{(def } R_<\text{)}\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I\rrbracket h + \mu_2\rrbracket\sigma_2 k && \text{(def } \mathcal{C}\llbracket\rrbracket\text{)}
\end{aligned}
$$

So, $\mathcal{P}(\theta', t', p+)$ holds.

thus completing the proof of our main theorem, and ensuring the consistency of the whole SSA conversion process. □

We are left with the simple issue of checking that state consistency is satisfied for the initial states.

LEMMA 4. *$\mathcal{P}(\bot, \bot, (1, 0^m))$ holds.*

PROOF.

$$
\begin{aligned}
&\mathcal{I}\llbracket I\rrbracket(1,0^m)\bot = \\
&= R_{<1.1}(\bot I) && \text{(def } \mathcal{I}\llbracket\rrbracket\text{)}\\
&= \bot \\
&= \mathcal{E}\llbracket R_{<1}(\bot I)\rrbracket\bot 0^m && \text{(def } R_<\text{)}\\
&= \mathcal{E}\llbracket \mathcal{C}\llbracket I\rrbracket 1\bot\rrbracket\bot 0^m && \text{(def } \mathcal{C}\llbracket\rrbracket\text{)}
\end{aligned}
$$

□

The final theorem wraps things up by showing that after evaluating an SSA-converted program from consistent initial states, we end up in states that remain consistent. Note that this remains true even if the whole program loops.

THEOREM 2. *Given $S \in \mathit{Stmt}$, with $\theta = \mathcal{C}\llbracket S\rrbracket 1\bot$ and $t = \mathcal{I}\llbracket S\rrbracket(1,0^m)\bot$, the property $\mathcal{P}(\theta, t, (2, 0^m))$ holds.*

PROOF. Trivial using Lemma 4 and Theorem 1. □

7. Discussion

Even though the initial purpose of our work is to provide a firm foundation to the use of SSA in modern compilers, our results also yield an interesting theoretical insight on the computational power of SSA.

7.1 Recursive Partial Functions Theory

The mathematical wording of the Consistency Property 1 underlines a key aspect of the SSA conversion process. While p only occurs on the left hand side of the consistency equality, the syntactic location h and the iteration space vector k are uncoupled in the right-hand side expression. Thus, via the SSA conversion process, the standard semantics gets staged, informally getting "curryied" from $\mathit{Stmt} \to (N^* \times N^m) \to T \to T$ to $\mathit{Stmt} \to N^* \to N^m \to T \to T$; this is also visible on Figure 4, where the pair (h, k) is used on the left arrow, while h and k occur separately on the top and on the right arrows. This perspective change is rather profound, since it uncouples syntactic sequencing from run time iteration space sequencing.

There exists a formal computing model that is particularly well suited to describing iteration behaviors, namely Kleene's theory of partial recursive functions [19]. In fact, the SSA appears to be a syntactic variant of such a formalism. We provide below a rewriting $\mathcal{K}\llbracket\rrbracket$ of SSA bindings to recursive function definitions.

First, to each SSA identifier I, we associate a function $I(k)$, and translate any SSA expression involving neither $\mathsf{loop}\phi$ nor $\mathsf{close}\phi$ nodes[5] as function calls:

$$
\begin{aligned}
\mathcal{K}\llbracket N\rrbracket k &= N\\
\mathcal{K}\llbracket I\rrbracket k &= I(k)\\
\mathcal{K}\llbracket E_1 \oplus E_2\rrbracket k &= \oplus(\mathcal{E}\llbracket E_1\rrbracket k, \mathcal{E}\llbracket E_2\rrbracket k)
\end{aligned}
$$

Then, to collect partial recursive function definitions corresponding to an SSA program σ, we simply gather all the definitions for each binding, $\bigcup_{I\in \mathit{Dom}\ \sigma} \mathcal{K}\llbracket I, \sigma I\rrbracket$. Informally, for $\mathsf{loop}\phi$ expressions, we simply rewrite the two cases corresponding to their standard semantics. For $\mathsf{close}\phi$ expressions, we add an ancillary function that computes the minimum value (if any) of the loop counter corresponding to the number of iterations required to compute the final value, and plug it into the final expression.

[5] Without loss of generality, we assume that ϕ nodes only occur as top-level expression constructors.

This is formally defined as follows, using $k_{p,q}$ as a shorthand for $k_p, k_{p+1}, ..., k_{q-1}, k_q$:

$$\begin{aligned}
&\mathcal{K}[\![I, \mathsf{loop}_\ell\phi(E_1, E_2)]\!]k = \\
&\quad \{I(k_{1,\ell-1}, 0, k_{\ell+1,m}) = \mathcal{K}[\![E_1]\!](k_{1,\ell-1}, 0, k_{\ell+1,m}), \\
&\quad\ \ I(k_{1,\ell-1}, x+1, k_{\ell+1,m}) = \mathcal{K}[\![E_2]\!](k_{1,\ell-1}, x, k_{\ell+1,m})\} \\
&\mathcal{K}[\![I, \mathsf{close}_\ell\phi(E_1, E_2)]\!]k = \\
&\quad \{\mathtt{min}_I(k_{1,\ell-1}, k_{\ell+1,m}) = \\
&\qquad\qquad (\mu y.\mathcal{K}[\![E_1]\!](k_{1,\ell-1}, y, k_{\ell+1,m}) = 0), \\
&\quad\ \ I(k) = \mathcal{K}[\![E_2]\!](k_{1,\ell-1}, \mathtt{min}_I(k_{1,\ell-1}, k_{\ell+1,m}), k_{\ell+1,m})\} \\
&\mathcal{K}[\![I, E]\!]k = \{I(k) = \mathcal{K}[\![E]\!]k\}
\end{aligned}$$

where μ is Kleene's minimization operator. We also assumed that boolean values are coded as integers (*false* is 0).

7.2 Example

As an example of this transformation to partial recursive functions, we provide below the translation of our running example (see Figure 5) into partial recursive functions. For increased readability, we renamed variables to use shorter indices.

$$\begin{aligned}
&\mathtt{I}_1(k_1) = 7 \\
&\mathtt{J}_1(k_1) = 0 \\
&\mathtt{J}_2(0) = \mathtt{J}_1(0) \\
&\mathtt{J}_2(x+1) = \mathtt{J}_3(x) \\
&\mathtt{J}_3(k_1) = +(\mathtt{J}_2(k_1), \mathtt{I}_1(k_1)) \\
&\mathtt{min}_{\mathtt{J}_4}() = (\mu y. < (\mathtt{J}_2(y), 10) = 0) \\
&\mathtt{J}_4(k_1) = \mathtt{J}_2(\mathtt{min}_{\mathtt{J}_4}())
\end{aligned}$$

Figure 5. Partial recursive functions example.

Our conversion process from Imp to SSA can thus be seen as a way of converting any RAM program [12] to a set of Kleene's partial recursive functions, thus providing a new proof of Turing's Equivalence Theorem between these two computational models, previously typically proven using simulation [12].

8. Future Work

We looked in this paper at the Imp-to-SSA conversion process. A natural dual problem of course arises, namely the so-called "out-of-SSA" [6, 4, 18] issue: a way of prettyprinting SSA programs using typical, imperative-like programming language syntax such as Imp. This is of utmost importance when one considers for instance the issues of debugging or code generation. In GCC, this is dealt with using a graph algorithm [7] operating on the control-flow data structure decorated with the SSA annotations used in its middle end.

For our approach, this technique could also be used in a similar fashion, assuming we kept around the control-flow graph from which our SSA code has been generated. A more intriguing question is whether such an out-of-SSA Imp code generator could be designed using only our self-contained SSA syntax. In a perfect world, one would indeed want to get back the original Imp code from which SSA has been generated. This requires reconstructing the while loop structure using data dependence within SSA code, together with an intelligent ordering of code generation for each binding in σ to minimize code duplication.

9. Conclusion

We presented the first denotational specifications of both the semantics of SSA and of its conversion process from a core imperative programming language. SSA is the central control-flow intermediate representation format used in the middle ends of modern compilers such as GCC or Intel CC that target multiple source languages. Yet, there is surprisingly very limited work studying the formal properties of this central data representation technique.

Our main theorem proves that standard semantics is preserved after the transformation of imperative programs to their SSA intermediate forms. As a by-product, it provides another reduction proof for the RAM computational model to Kleene's partial recursive functions theory.

Since our results ensure the correctness of the translation process of all imperative programs to SSA, they pave the way to additional research from the programming language community, for instance for optimization purposes, which would directly target SSA instead of source languages. Using SSA as the language of interest for such endeavors would ensure the portability of the resulting algorithms (see [1] for some examples) to all programming languages supported by GCC or other similar compilers. This applies to both imperative or object-oriented programming languages (such as C or Java via GCC) or functional ones (such as Erlang via HiPE [15]).

Acknowledgments

The authors thank Neil Jones for his help regarding Kleene's partial recursive functions theory, Ken Zadeck for his remarks on closeϕ nodes and François Irigoin for his suggestions.

References

[1] A. W. Appel. *Modern Compiler Implementation*. Cambridge University Press, 1998.

[2] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998.

[3] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[4] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[7] GCC implementation of "out of SSA". `http://gcc.gnu.org/viewcvs/trunk/gcc/tree-outof-ssa.c`.

[8] The GNU Compiler Collection. `http://gcc.gnu.org`.

[9] S. Glesner. An ASM semantics for SSA intermediate representations. In *Proceedings of the 11th International Workshop on Abstract State Machines*. Springer Verlag, Lecture Notes in Computer Science, May 2004.

[10] M. J. C. Gordon. *The denotational description of programming languages*. Springer Verlag, 1979.

[11] Intel compilers. `http://intel.com/`.

[12] N. D. Jones. *Computability and complexity: from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997.

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

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Conf. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[15] D. Luna, M. Pettersson, and K. Sagonas. Efficiently compiling a functional language on AMD64: the HiPE experience. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 176–186, New York, NY, USA, 2005. ACM Press.

[16] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[17] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 218–232, Barcelona, Spain, Nov. 2005. Springer-Verlag.

[18] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag.

[19] J. E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Languages Theory*. MIT Press, 1977.

[20] F. K. Zadeck. Loop closed SSA form. Personal communication.

[21] F. K. Zadeck. Static single assignment form, 2004 GCC Summit keynote. `http://naturalbridge.com/GCC2004Summit.pdf`.