HAL
archives-ouvertes.fr

In and Out of SSA : a Denotational Specification

Sebastian Pop, Pierre Jouvelot, George André Silber

▶ **To cite this version:**

Sebastian Pop, Pierre Jouvelot, George André Silber. In and Out of SSA : a Denotational Specification. Workshop Static Single-Assignment Form Seminar„ Apr 2009, Autrans, France. hal-00915979

HAL Id: hal-00915979

https://hal-mines-paristech.archives-ouvertes.fr/hal-00915979

Submitted on 9 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

In and Out of SSA: A Denotational Specification

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

Open Source Compiler Engineering, Advanced Micro Devices Inc., Austin, Texas,
[†]CRI, École des mines de Paris, France
sebastian.pop@amd.com, {jouvelot, silber}@cri.ensmp.fr

Abstract

We present non-standard denotational specifications of the SSA form and of its conversion processes from and to imperative programming languages. Thus, we provide a strong mathematical foundation for this intermediate code representation language used in modern compilers such as GCC or Intel CC.

More specifically, we provide (1) a new functional approach to SSA, the Static Single Assignment form, together with its denotational semantics, (2) a collecting denotational semantics for a simple imperative language Imp, (3) a non-standard denotational semantics specifying the conversion of Imp to SSA and (4) a non-standard denotational semantics for the reverse SSA to Imp conversion process. These translations are proven correct, ensuring 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 unexpected by-products of our conversion procedures, we offer (1) 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, and, on a more practical note, (2) a new algorithm to perform program slicing in imperative programming languages. All these specifications have been prototyped using GNU Common Lisp.

These fundamental results prove 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 analysis techniques 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, RAM model, partial recursive functions theory, program slicing

1. Introduction

Many modern and widely distributed compilers for imperative and even some functional languages use the SSA form as an intermediate code representation formalism. The Static Single Assignment (SSA) form [29] 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 traditionally is graph-based, and represents basic blocks linked within a control-flow graph. When more than one path reach 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.

Based on simple concepts, SSA is surprisingly efficient; various compiler optimization algorithms such as constant propagation or dead-code elimination are of lower complexity when specified on SSA than when tuned to more classical control-flow graphs (see [36]). This formalism has therefore been widely used in both academic (e.g., GCC [18, 30], LLVM [26]) and commercial (Intel CC [33]) compilers.

Yet, we believe the theoretical foundations of SSA are somewhat lacking (see Section 2 for a presentation of 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 processes used to translate imperative source code to intermediate SSA constructs and back. Our work intends then to strengthen the core formalism of SSA and enable the introduction of more formal correctness proofs for SSA-based optimization algorithms in the future, a key concern given the importance of code correctness in software engineering tools as crucial as compilers.

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 reaching the end of the alternative branches. Thus, most of the methods of ϕ-node placement present in the literature, when dealing with the related but, we believe, somewhat different ϕ-nodes that logically occur after structured loops, simply consider them as equivalent to conditional ϕ-nodes. In particular, loop exit conditions are not traditionally considered an intrinsic part of the SSA; in practice this is not a major issue, since most compilers' middle ends keep code information on the side (e.g., control-flow graphs or continuations) from which they can be retrieved.

The introduction of loop-specific ϕ-nodes by the research community (see [35, 5]) with the "Gated SSA" variant of SSA, in which such expressions do appear in loop nodes, was mostly motivated by the desire to extend SSA to dataflow languages and architectures,

while its recent adoption by the GCC community [14, 39, 38] is practical and related to the ease with which code transformation algorithms working directly on loop structures and not general graphs can be designed. These somewhat narrow-focused beginnings may explain why loop-specific ϕ-nodes were overlooked in recent surveys of SSA [6], as their semantic role was not quite 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 Imp, a very basic yet complete imperative programming language, and provide its standard denotational semantics. In Section 4, we formally present our functional definition of 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 proofs. 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; we also provide our first theorem, which shows that Imp and SSA evaluation processes preserve the consistency of memory states. We look at the dual issue of SSA-to-Imp conversion in Section 6, which includes its specification and our second correctness proof. Building on these core results, we discuss in Section 7 some consequences of our results, in particular the reduction of RAM programs to partial recursive functions and the application of our conversion processes to program slicing. We look at future work in Section 8 and conclude in Section 9. All proofs can be found in [31].

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 [11, 12], which introduce the notation for SSA, mostly present informal semantics and proofs for some optimization algorithms based on the SSA representation.
- Kelsey [24] 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 [4]). 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 and vice versa. In addition, we provide mathematical correctness proofs for these conversion processes.
- A similar semantics, based on continuations, is given by Glesner [19]: 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 processes for imperative programs.
- A rule-based operational semantics for a graph version of SSA, an algorithm for translating statements expressed in a register-level language into SSA and a somewhat informal correctness proof of this process are given in [37]. A survey of optimizations based on SSA is also provided. Our approach focuses on a new, functional syntax and semantics for SSA, and offers formal proofs of its direct translation from and to a high-level structured imperative language.

The already mentioned work of Ballance and al. [5], perhaps not coincidentally mostly targeted to the extension of the SSA form to the dataflow (and hence functional) computing paradigm, offers some striking similarities to our approach of SSA and its semantics. The authors introduce the "Program Dependence Web", built on top of a variant of SSA, the "Gated SSA", GSA, which introduces specific loop nodes in the SSA representation, as we do. Our results indirectly provide a simplification of their definition (GSA is a graph-based representation with three gating functions, while we show that only two, graph-independent constructs are, in fact, needed), a clean denotational semantics for SSA and direct translations between imperative programs and SSA (the GSA conversion algorithms are built on top of the Program Dependence Graph of Ferrante and al [17]). Finally, we provide formal correctness proofs for our denotationally-based transformations and show that SSA has the computational power of the RAM computing model.

Indeed, beside looking at the formal definition of SSA semantics, our paper also addresses the issues of converting SSA from and to imperative programs, which implies that a proper framework for specifying these processes be used. The usual references in the literature [8, 12, 7, 5] rest on graph algorithms, which is appropriate given the graph nature of classical SSA. Since we take here a purely programming language-based approach, we use a different theoretical foundation to both express these processes and prove their correctness. We found the denotational framework [34] to be well suited to this task, given the structural definitions of our languages and our desire to express precise, formal correctness proofs, required to ensure the soundness of SSA; all the specifications we provide below can be seen as non-standard denotational semantics [23, 13] of imperative or SSA programs defined on the abstract syntax of the languages under study.

An added, practical benefit with this approach is that such specifications are executable using any functional language [23]; since we were only interested here in getting a proof-of-concept implementation, we used GNU Common Lisp as our "executable specification language" [20].

3. Imp, the Simple Imperative Language

Since we are interested in this paper by the basic principles underpinning the SSA conversion processes, we use a very simple yet RAM-complete imperative language, Imp, based on assignments, sequences and while loops[1]. As is well-known, conditional statements can always be encoded by a sequence of one or two loops [32], and thus need not be part of our core syntax.

[1] The RAM model requires array-like constructs that are missing from our definition. Since SSA mostly deals with control structures, we don't see this as a restriction, since adding such aggregate data structures is a dual issue to the ones we tackle in this paper.

3.1 Syntax

Imp is defined by the following syntax:

$$\begin{array}{rcl} N & \in & Cst \\ I & \in & Ide \\ E & \in & \mathit{Expr} ::= N \mid I \mid E_0 \oplus E_1 \\ S & \in & \mathit{Stmt} ::= I := E \mid S_0; S_1 \mid \mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{end} \end{array}$$

with the usual predefined integer constants, identifiers and operators $\oplus$.

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 execution. To keep track of such execution points, we use both syntactic and iteration space information:

- Each statement in the program tree is identified by a Dewey 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; it is used to identify the x-th son of the node located at h in the program tree. 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 located at $h+$, which is defined as follows, assuming tree nodes with m children:

$$\begin{array}{rcl} n+ & = & n+1 \\ (h.n)+ & = & h.(n+1) \quad (1 \le n < m) \\ (h.m)+ & = & h+ \end{array}$$

- To deal with the distinct iterations of program loops, we use iteration space functions k, of type $K = N^* \to N$. The value kh of Function k for a while statement located at h denotes the current loop index value for this loop. Informally, k collects the counter values for all loops (identified by their Dewey number); during execution, this function is updated as loops unroll, and we note $k[a/h]$ the function obtained from k by replacing the value at Index h with a[2].

To sum up, a program execution point p is a pair $(h,k) \in P = N^* \times K$ 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. Intuitively, each (h,k) occurs only once in a given execution trace (the ordered sequence of all states).

The only requirement on points is that they be lexicographically ordered, with the infix relation $< \in P \times P \to \mathit{Bool}$ such that $(h_1,k_1) < (h,k) = (k_1 < k \vee (k_1 = k \wedge h_1 < h))$; the order relationship $<$ on iteration functions k is straightforwardly defined over their ordered domains.

3.2 Semantics

As usual, the denotational semantics of Imp operates upon functions f on lattices or CPOs [34]; all the domains we use thus have a $\perp$ minimum element. The definition domain of f, i.e., the set of values on which it is defined, is given as $\mathit{Dom}\ f = \{x \mid f(x) \neq \perp\}$.

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

$$\begin{array}{rcl} \mathcal{I}[\![N]\!]pt & = & in_V(N) \\ \mathcal{I}[\![I]\!]pt & = & R_{<p}(tI) \\ \mathcal{I}[\![E_0 \oplus E_1]\!]pt & = & \mathcal{I}[\![E_0]\!]pt \oplus \mathcal{I}[\![E_1]\!]pt \end{array}$$

where the only unusual aspect of this definition[3] is the use of $R_{<x}f = f(\max_{<x} \mathit{Dom}\ f)$, 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 at which I has been updated, prior the current p; since we use a collecting semantics, we need to "search" the states to find this last definition.

To specify the semantics of statements, we need to introduce augmented states $u \in U = K \times T$, called "rolling states", that combine iteration space functions and states. The semantics of statements $\mathcal{I}[\![\,]\!] \in \mathit{Stmt} \to N^* \to U \to U$ yields the rolling state obtained after executing the given statement at the given program Dewey number, given an incoming state $u = (k,t)$:

$$\begin{array}{rcl} \mathcal{I}[\![I := E]\!]h(k,t) & = & (k, t[\mathcal{I}[\![E]\!](h,k)t/(h,k)/I]) \\ \mathcal{I}[\![S_0; S_1]\!]h & = & \mathcal{I}[\![S_1]\!]h.2 \circ \mathcal{I}[\![S_0]\!]h.1 \end{array}$$

These definitions are rather straightforward extensions of a traditional standard semantics to a collecting one. For an assignment, we add a new binding of Identifier I at Point (h,k) to the value of E. A sequence simply composes the transformers associated to S_0 and S_1 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 $\mathrm{fix}(W_h)$ of the W_h functional defined as:

$$\mathcal{I}[\![\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{end}]\!]h(k,t) = \mathrm{fix}(W_h)(k[0/h],t)$$

$$W_h = \lambda w.\lambda u.\ \begin{cases} w(k'_{h+},t'), & \text{if } \mathcal{I}[\![E]\!](h.1,k)t, \\ u, & \text{otherwise.} \end{cases}$$

$$\text{where } (k',t') = \mathcal{I}[\![S]\!]h.1 \text{ and } (k,t) = u$$

where, as a shorthand, k_{h+} is the same as k, except that the value at Index h is incremented by one (similarly, we latter use k_{h-}, with a decrement by one).

Beginning with an iteration vector set to 0 for Index h, if the value of the guarding expression E is true, we iterate the while loop with a state updated by the loop body, while incrementing the iteration space vector, since an additional loop iteration has taken place. If the loop test is false, we simply consider the loop as a no-op.

3.3 Example

To illustrate our results, we use a single example running throughout this paper; we provide in Figure 1 this very 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. Since we implemented all the denotational specifications provided in this paper in GNU Common Lisp, interested readers are welcome to try longer examples using this prototype [22].

In this example, if we assume that the whole program is at Dewey number 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, the iteration space function domain has only one element, at Dewey number 1.2.2. Thus, for instance, after two loop iterations, the value of J is 14, and this will cause the loop to

[2] 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 extend f at a given value x.

[3] 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).

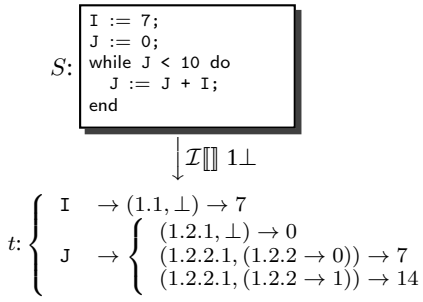

Figure 1. Syntax and semantics for an Imp program: $(k, t) = \mathcal{I}[\![S]\!]1\bot$.

terminate. The collecting nature of the semantics is exemplified here by the fact that we keep track of all values assigned to each variable throughout the whole computation.

4. SSA

If the definition of Imp given above is rather straightforward, the treatment of SSA given below is new. We motivate in this section the need for such a fresh approach to SSA and specify its syntax and semantics.

4.1 Functional SSA

In the standard SSA terminology [12, 29], the SSA intermediate representation of an imperative program is a graph of def-use chains in SSA form. Nodes in the graph are basic blocks possibly ending with a test, and each assignment targets a unique variable. ϕ nodes occur at merge points of the control flow graph to restore the proper values from the renamed variables according to the semantics of the original imperative constructs (i.e., to represent the proper evolution of variables in loop and conditional statements). As an example, Figure 2 provides the graph-based SSA representation for our running example given in Figure 1.

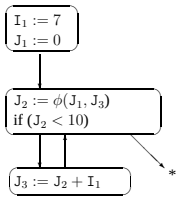

Figure 2. Classical SSA graph.

This original representation of SSA suffers from one drawback: variable names defined in loops are accessible from anywhere after the loop. For instance, one could, and indeed sometimes does to get the exit value of Variable J, write "`X := J_2`" in a basic block that follows the starred exit arc of the while graph. Although operationally valid, such accesses clearly lack structure, since all accesses to J from the SSA representation need to go deep into the graph structure. Moreover, multiple exit arcs can be a problem when dealing with SSA graph operations such as insertion and deletion of edges.

The current versions of GCC, beginning with Version 4.0, use "loop close" ϕ nodes [14] that are inserted immediately after the loop for each variable used outside of the loop. This ensures that every edge of the SSA graph points to a variable defined at most one loop level deeper, avoiding complicated SSA graph rewiring after code transformations. The "loop closed" version of the SSA graph for our running example can be seen in Figure 3.

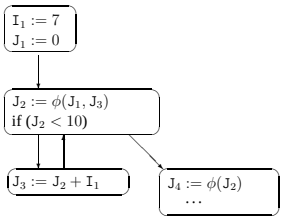

Figure 3. Loop closed SSA graph.

Even though this representation is admittedly more structured than the original one, maintaining a proper control-flow graph on the side of these SSA expressions is still required, if only to grant access to the exit boolean expressions that label while loops.

In this paper, we suggest to go one step further by recognizing that one can replace this whole graph-based approach with a programming language-based paradigm. In this new "functional SSA" form, the ϕ assignments are capturing all the control characteristics of programs, making usual control-flow primitives consequently redundant. The corresponding functional SSA code for our running example is given in the upper part of Figure 4 (see next subsections for a formal explanation of our syntax and semantics).

The definition of this self-contained, functional format for SSA is one of the new ideas we introduce in this paper. This programming language approach provides a more formal view of the definition of SSA, its conversion processes and their correctness; standard yet powerful proof techniques developed in the realm of programming language theory can, as shown below, be more readily applied here than when using graph-based representations.

4.2 Syntax

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

$$E \in SSA ::= N \mid I_h \mid E_0 \oplus E_1 \mid \mathsf{loop}_h(E_0, E_1) \mid \mathsf{close}_h(E_0, E_1)$$

which extend the basic imperative definitions of *Expr* with two types of ϕ expressions: loop and close terms. ϕ nodes that merge expressions declared at different loop depths are called loop_h nodes and have a recursive semantics. A close_h node collects the final value that comes either from the loop h or from before the loop h, when the loop trip count, related to the first argument, is zero. Since we stated that imperative control flow primitives should not be part of our SSA representation, we intendedly annotate ϕ nodes with a label information h 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.

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 imperative conditional statements, which without loss of generality are, as mentioned above, absent from the syntax of Imp; these nodes would be handled by a proper combination of loop and close nodes.

Note that identifiers I_h in an SSA expression are also labeled with a Dewey number. Since every assignment in Imp is located at a unique h, we use, in the Imp-to-SSA conversion process described below, this number to uniquely tag identifiers in order to ensure that no identifiers in an imperative program will ever be assigned twice once converted to SSA form, thus enforcing its static single assignment property.

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

4.3 Semantics

Since in an SSA program σ all expressions in its image domain recursively refer, via identifiers, to the same σ, the semantics of σ uses an environment $\rho \in H = \mathit{Ideh} \to K \to V$, defined as a fixed point of the environment extension function $\mathcal{R} \in (\mathit{Ideh} \times SSA) \to \rho \to \rho$, which iterates over the domain of σ. The semantics function for SSA expressions $\mathcal{E}[\![]\!]$ has then type $SSA \to H \to K \to V$; it associates to a given expression in such a recursively constructed environment and with an iteration space function its value.

The semantics of an SSA program σ is thus the finite function $\mathcal{R}\sigma$ defined as follows:

$$\begin{aligned} \mathcal{R}\sigma &= \mathrm{fix}(\bigsqcup_{I \in Dom\ \sigma} \mathcal{R}(I, \sigma I)) \\ \mathcal{R}(I,E)\rho &= \rho[\lambda k.\mathcal{E}[\![E]\!]\rho k/I] \end{aligned}$$

where $\mathcal{R}$ is used, via the fixed point operator, to build a recursive environment ρ in which all identifiers I, when given an iteration function k, are bound to the value of the expression $E = \sigma I$ that defines them in σ. The evaluation of such an expression is defined below:

$$\begin{aligned} \mathcal{E}[\![N]\!]\rho k &= in_V(N) \\ \mathcal{E}[\![I]\!]\rho k &= \rho I k \\ \mathcal{E}[\![E_0 \oplus E_1]\!]\rho k &= \mathcal{E}[\![E_0]\!]\rho k \oplus \mathcal{E}[\![E_1]\!]\rho k \\ \mathcal{E}[\![\mathsf{loop}_h(E_0,E_1)]\!]\rho k &= \begin{cases} \mathcal{E}[\![E_0]\!]\rho k, \text{if } kh = 0, \\ \mathcal{E}[\![E_1]\!]\rho k_{h-}, \text{otherwise.} \end{cases} \\ \mathcal{E}[\![\mathsf{close}_h(E_0,E_1)]\!]\rho k &= \\ & \mathcal{E}[\![E_1]\!]\rho k[\min\{x \mid \neg\mathcal{E}[\![E_0]\!]\rho k[x/h]\}/h] \end{aligned}$$

Constants such as N are denoted by themselves. We already explained how the semantics of identifiers relies on the recursively built environment ρ. Operator-based expressions are straightforwardly defined by induction.

loop_h nodes, by their very iterative nature, are designed to represent the values of variables successively modified in imperative loop bodies, while close_h nodes compute the final value of such induction variables in loops guarded by test expressions related to E_0. Of course, when a loop is infinite, there is no iteration that exits the loop, i.e., there is no k such that $\neg\mathcal{E}[\![E_0]\!]\rho k$, and thus the set $\{x \mid \neg\mathcal{E}[\![E_0]\!]\rho k[x/h]\}$ is empty. In such a case, $\min\emptyset$ corresponds to $\bot$.

4.4 Example

We informally illustrate in Figure 4 the semantics of SSA using an SSA program σ 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 iteration vectors to a constant. To merge the two paths reaching in Imp the loop body, we use a loop expression to combine the initial value of J and its successive iterated values within the loop. A close expression "closes" the iterative function associated to $\mathtt{J}_2$ to retrieve its final value, obtained when the test expression evaluates to *false*; in this case, this yields 14, if evaluated in $\mathcal{R}\sigma$.

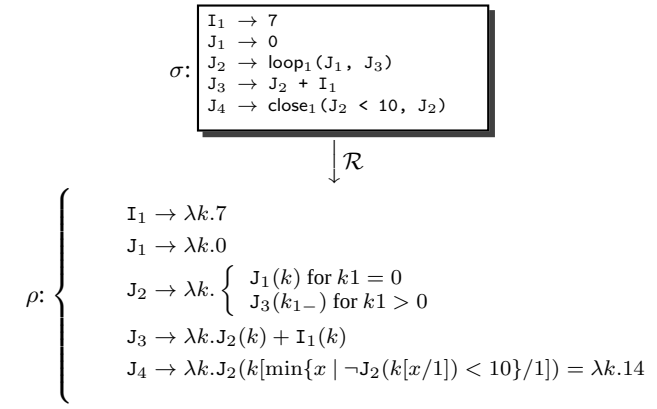

Figure 4. Syntax and semantics of ϕ expressions: $\rho = \mathcal{R}\sigma$.

5. Conversion of Imp 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: $\mu \in M = Ide \to N^* \to \mathit{Ideh}$ 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); $\sigma \in \Sigma = \mathit{Ideh} \to SSA$ simply collects the SSA definitions associated to each identifier in the image of M.

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

$$\begin{aligned} \mathcal{C}[\![N]\!]h\mu &= N \\ \mathcal{C}[\![I]\!]h\mu &= R_{<h}(\mu I) \\ \mathcal{C}[\![E_0 \oplus E_1]\!]h\mu &= \mathcal{C}[\![E_0]\!]h\mu \oplus \mathcal{C}[\![E_1]\!]h\mu \end{aligned}$$

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

$$\begin{aligned} \mathcal{C}[\![S_0; S_1]\!]h &= \mathcal{C}[\![S_1]\!]h.2 \circ \mathcal{C}[\![S_0]\!]h.1 \\ \mathcal{C}[\![I := E]\!]h(\mu,\sigma) &= (\mu[I_h/h/I], \sigma[\mathcal{C}[\![E]\!]h\mu/I_h]) \end{aligned}$$

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:

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

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 are bound to loop 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 loop 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 close nodes that represent their values when the loop exits (or $\bot$ if the loop is infinite, as we will see). All references to any identifier once the loop is performed are references to these close expressions located at $h.2$, which follows, by definition of the lexicographic order on points, all other points present in the loop body.

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

5.2 Example

We find in Figure 5 the result of the Imp-to-SSA conversion algorithm on our running example: $\mathcal{C}[\![S]\!]1\bot$. The SSA code σ, i.e., a mapping of SSA identifiers to SSA expressions, represented here as a tabulated list, is taken verbatim from the output of our GNU Common Lisp prototype. The current binding for the imperative identifier `I` in μ is `I_12212`, and `J_12212` for `J` (we represent Dewey numbers in SSA identifiers as suffixes preceded by `_`; the dots are removed for readability purposes).

As expected, this SSA program is similar to the one in Figure 4, up to the renaming of the SSA identifiers. Both `I` and `J` are bound to close expressions with the same test expression that involves the value of `J_12210`, a loop expression that evaluates to 0 for the first iteration and to the sum of `J+I` translated in SSA form for the subsequent ones. Note how the value of the loop invariant `I` is managed by the loop expression `I_12210` which evaluates to 7 for the first iteration, and keeps its value afterwards; a simple SSA code optimization[4] would replace this binding in σ with (`I_12210, I_11`), equivalent modulo termination to the constant binding in Figure 4.

As advertised earlier, 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.

[4] Proving the correctness of this optimization could be performed using the denotational semantics of SSA provided in this paper. More generally, the need for optimization in SSA-generated code is discussed in Section 8.

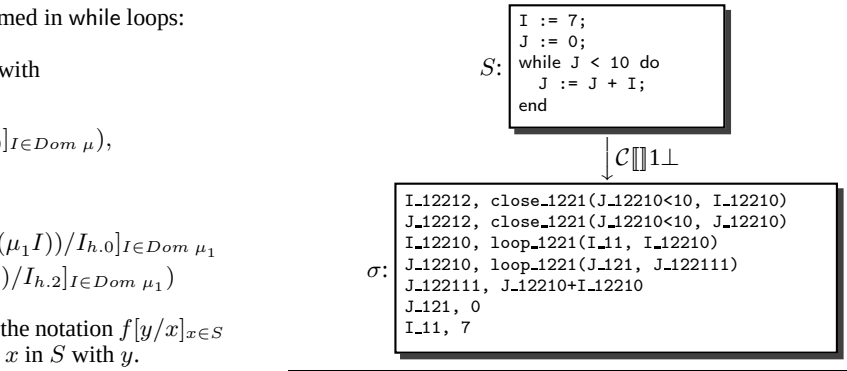

Figure 5. Conversion from Imp to SSA: $(\mu, \sigma) = \mathcal{C}[\![S]\!]1\bot$.

5.3 SSA Conversion Consistency

We are finally equipped with all the material required to express our first 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 $P(\theta, t, p)$, iff*

$$\forall I \in Dom\ t, \mathcal{I}[\![I]\!]pt = \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!](\mathcal{R}\sigma)k \tag{1}$$

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 6).

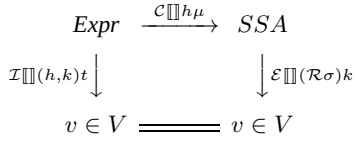

Figure 6. Consistency property $P((\mu, \sigma), t, (h, k))$.

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

LEMMA 1 (Consistency of Expression Conversion). *Given that $P(\theta, t, p)$ with $p = (h, k)$, and an expression $E \in$ Expr,*

$$\mathcal{I}[\![E]\!]pt = \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!](\mathcal{R}\sigma)k \tag{2}$$

This directly leads to our main theorem, which ensures the semantic correctness of the conversion process from imperative constructs to SSA expressions:

THEOREM 1 (Consistency of Statement Conversion). *Given any statement S and for all $\theta, t, p = (h, k)$ that verify $P(\theta, t, p)$, then $P(\theta', t', (h+, k'))$ holds with*

$$\begin{aligned}
\theta' &= \mathcal{C}[\![S]\!]h\theta\\
(k', t') &= \mathcal{I}[\![S]\!]h(k, t)
\end{aligned}$$

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.

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

LEMMA 2. *$P(\bot, \bot, (1, \lambda h.0))$ holds.*

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

THEOREM 2. *Given $S \in$ Stmt, with $\theta = \mathcal{C}[\![S]\!]1\bot$ and $(k, t) = \mathcal{I}[\![S]\!]1(\lambda h.0, \bot)$, the property $P(\theta, t, (2, k))$ holds.*

PROOF. Trivial using Lemma 2 and Theorem 1. □

6. Conversion of SSA to Imp

If the functional characteristics of SSA makes it particularly well suited to program optimizations (see e.g., [25]), for such optimized programs to run one has to find a way to get back to the imperative paradigm required by most current computer architectures. We provide in this section an algorithm that translates an SSA program to its Imp equivalent, and explicit its correctness.

6.1 Specification

An SSA program σ specifies a binding of identifiers to expressions. Thus, getting an imperative version for such a program amounts to discovering the Imp code required to compute the value of each "useful" identifier. In the framework of compiler middle-ends we envision in this paper, these will in fact be the identifiers required to compute the result of imperative programs that were translated into σ. Our SSA-to-Imp core translation function $\mathcal{O}[\![]\!]$ takes thus as first argument one of these bindings, i.e., an identifier I and an SSA expression E, and returns the Imp code required to compute "$I := E$".

Since SSA expressions include loop constructs which, per se, do not directly correspond to actual Imp code (i.e., enclosing close expressions are required to specify loop bounds), we introduce "loop environment" functions to keep track of pieces of Imp code that eventually will be used to generate the whole imperative program. The domain of a loop environment is a set of "loop aspects", which are tuples $a = (h, b) \in A$, with h a Dewey number and $b \in Y = \{\mathsf{head}, \mathsf{body}, \mathsf{iter}, \mathsf{env}\}$ a symbol.

A loop environment $\kappa \in L = A \to (\mathit{Stmt} + P(Ide \times Y))$ maps such aspects to the statements associated to either the header, the body or the iteration step of the loop designated by the Dewey number h. A loop characterized by its Dewey number h can thus be seen as the following pattern:

$\kappa(h, \mathsf{head})$;
while <test_expression>
  $\kappa(h, \mathsf{body})$;
  $\kappa(h, \mathsf{iter})$;
end

Beyond these "structural" aspects, κ also maps "environment" aspects (with the symbol env) to the sets of identifiers defined in Loop h; these sets are key in both the specification of the conversion process and the correctness proof. We use two helper functions to maintain loop environments:

$$\begin{aligned} \mathsf{up}[\![I, S]\!]a\kappa &= \mathsf{up}_{\mathsf{env}}[\![I]\!]a(\kappa[\kappa a; S/a]) \\ \mathsf{up}_{\mathsf{env}}[\![I]\!](h, b)\kappa &= \kappa[\kappa(h, \mathsf{env}) \cup \{(I, b)\}/(h, \mathsf{env})] \end{aligned}$$

where $\mathsf{up}[\![]\!]$ extends with the statement S the code of Aspect a which computes the value of I while also updating the environment of Loop h with the newly defined identifier, via a call to $\mathsf{up}_{\mathsf{env}}[\![]\!]$.

The out of SSA conversion specification is $\mathcal{O}[\![]\!] \in (\mathit{Ideh} \times SSA) \to \Sigma \to A \to L \to L$. The term $\mathcal{O}[\![I, E]\!]\sigma a\kappa$ is an extended loop environment in which the Imp code required to assign the imperative equivalent of E to I is added to the code of Aspect a in Loop environment κ, using σ to find the definitions of the free SSA variables that are required to evaluate E. It is defined, for arithmetic expression bindings, as follows:

$$\begin{aligned} &\mathcal{O}[\![I, N]\!]\sigma = \mathsf{up}[\![I, I := N]\!], \\ &\mathcal{O}[\![I, I']\!]\sigma a\kappa = \mathsf{up}[\![I, I := I']\!]a\kappa_0, \text{with} \\ &\quad \kappa_0 = \begin{cases} \mathcal{O}[\![I', \sigma I']\!]\sigma a(\mathsf{up}_{\mathsf{env}}[\![I']\!]a\kappa), & \text{if } I' \notin \mathsf{dom}_{\mathsf{env}}(\kappa) \\ \kappa, \text{otherwise,} \end{cases} \\ &\mathcal{O}[\![I, E_0 \oplus E_1]\!]\sigma a\kappa = \mathsf{up}[\![I, I := I_0 \oplus I_1]\!]a\kappa_1, \text{with} \\ &\quad \kappa_1 = (\mathcal{O}[\![I_1, E_1]\!]\sigma a \circ \mathcal{O}[\![I_0, E_0]\!]\sigma a)\kappa \end{aligned}$$

where I_0 and I_1 denote fresh imperative variables, and $\mathsf{dom}_{\mathsf{env}}(\kappa) = \pi_1(\bigcup_{(h,\mathsf{env}) \in Dom\ \kappa} \kappa(h, \mathsf{env}))$ is the set of identifiers[5] of all the environment aspects of κ.

The case for constants is simple: we update the code for the corresponding aspect with the obvious assignment. This trivial case occurs again when dealing with identifier assignments if the defining identifier I' is already present in the environment. Otherwise, one needs, before assigning to I, to collect the code for the expression $\sigma I'$ that defines I'; this is done in a loop environment properly updated to reflect the fact that, since we are currently defining the code for I', there is no need to recurse if I' ever occurs again in the subsequent recursive calls to $\mathcal{O}[\![]\!]$.

Finally, the case for an operator uses straightforward recursive calls.

We focus now on "control-level" expressions in SSA:

$$\begin{aligned} &\mathcal{O}[\![I, \mathsf{loop}_h(E_0, E_1)]\!]\sigma a\kappa = \mathsf{up}[\![I, I := I_1]\!](h, \mathsf{iter})\kappa_1, \text{with} \\ &\quad \kappa_1 = (\mathcal{O}[\![I_1, E_1]\!]\sigma(h, \mathsf{body}) \circ \mathcal{O}[\![I, E_0]\!]\sigma(h, \mathsf{head}))\kappa, \\ &\mathcal{O}[\![I, \mathsf{close}_h(E_0, E_1)]\!]\sigma a\kappa = \mathsf{up}[\![I, W; I := I_1]\!]a\kappa_1, \text{with} \\ &\quad W = \kappa_1(h, \mathsf{head}); \mathsf{while}\ I_0\ \mathsf{do}\ \kappa_1(h, \mathsf{body}); \kappa_1(h, \mathsf{iter})\ \mathsf{end} \\ &\quad \kappa_1 = (\mathcal{O}[\![I_1, \mathsf{loop}_h(E_1, E_1)]\!]\sigma a \circ \mathcal{O}[\![I_0, \mathsf{loop}_h(E_0, E_0)]\!]\sigma a)\kappa \end{aligned}$$

As already alluded to, there is no stand-alone code generated for a loop expression; the aspect argument a, where this code is supposed to be added, is thus not used. Instead, one needs to distribute parts of the corresponding code in the various aspects of Loop h impacted by the loop_h expression: E_0 goes into the header and E_1 in the body. Note that the expression E_1 defining the new value I_1 of I may possibly refer to the old value of I obtained at the end of the previous loop iteration; I gets its new value I_1 in the code for the iteration aspect (h, iter) of the loop.

The code gathered in loop expressions is actually used when a close expression is encountered. The values of the loop test and body are bound to I_0 and I_1. We use loop expressions to compute these values both in the header and the body of the loop; these loop expressions are required since we may or may not enter the loop body and yet be able to provide a meaningful exit value to the rest of the program. The final value of I is obtained by an assignment of I_1, after the inclusion of Code W; this all-important code collects all code fragments relevant to the header, body and iteration step of Loop h and stored in the loop environment κ.

6.2 Example

To get a better grasp of the way our conversion executable specification works, we provide in Figure 7 the "out-of-SSA" Imp code for the original running SSA example given in Figure 4. This code is taken from the output of our GNU Common Lisp prototype, when requesting that the value of $\mathtt{J}_4$ be stored in the fresh variable RESULT (see Theorem 4 for details): $\mathcal{O}[\![\mathtt{RESULT}, \mathtt{J}_4]\!]\sigma(0, \mathsf{head})\bot$.

[5] π_1 is the first projector for pairs, here naturally extended to sets of pairs.

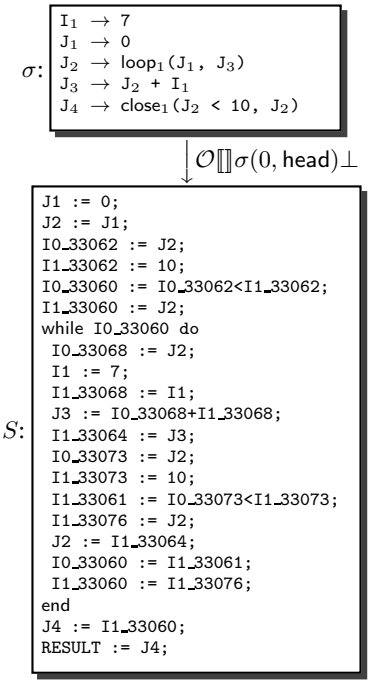

Figure 7. Conversion from SSA to Imp: $S = (\mathcal{O}[\![\texttt{RESULT}, \texttt{J}_4]\!]\sigma(0, \mathsf{head})\bot)(0, \mathsf{head})$.

In our Lisp implementation, fresh identifier names required by the specification are implemented by calls to Lisp's `gensym`, using `I0` and `I1` as string prefixes when called for by the specification.

Beside the obvious need for the straightforward code optimizations that we discuss in Section 8, it is easy to recognize from the code provided the basic control and data structures we were expecting. The various loop aspects - header, body and iteration - are also obvious to spot. Interestingly, except for the identifiers bound to SSA loop expressions, the imperative code also uses single assignment.

One slight but key difference between this code and the original imperative version in Figure 1 we were implicitly expecting to obtain is the position of the constant assignment to `I1`. The $\mathcal{O}[\![]\!]$ conversion process proceeds in a demand-driven manner, thus introducing the assignment to `I` only where it is actually needed. This is akin to the notion of "program slicing", an issue we address in Section 7.

6.3 Imp Conversion Consistency

As is the case for the Imp to SSA conversion process, we need to ensure that our reverse translation specification is correct. However, recalling that some SSA expressions such as loops do not individually directly lead to executable code, we need to take a somewhat indirect route to express our correctness requirement: indeed, we use whole programs to define the consistency property between imperative and SSA codes.

DEFINITION 2 (Out of SSA Consistency). *An SSA definition σ is* consistent *with a loop environment κ, noted $Q(\sigma, \kappa)$, iff for all $(h, \mathsf{env}) \in Dom\ \kappa$, Iteration Function k and State t, one has:*

$$\forall I \in \kappa(h, \mathsf{env}), \mathcal{E}[\![I]\!](\mathcal{R}\sigma)k = \mathcal{I}[\![I]\!](h+, k')t'$$

with

$$(k', t') = \mathcal{I}[\![\kappa(h, \mathsf{head}); (\kappa(h, \mathsf{body}); \kappa(h, \mathsf{iter}))^{kh}]\!]h(k, t)$$

where S^n is a shorthand for the sequence $S; \ldots; S$, where S is copied n times. Q basically states that the values of all identifiers I in Loop environment κ are identical whether they are considered as (1) SSA expressions in σ, using any loop iteration function k, or (2) Imp expressions, using the state one obtains after executing, from the initial state (k, t), the head and kh iterations of the loop body and iteration codes for Loop h.

Our second major theorem builds on this property by ensuring that all code extensions to κ introduced by calls to $\mathcal{O}[\![]\!]$ maintain this equivalence relation between SSA and Imp codes.

THEOREM 3 (Out of SSA Expression Consistency). *Given $Q(\sigma, \kappa)$, for any $I \notin \mathsf{dom_{env}}(\kappa)$, $E \in$ SSA, Dewey number h and structural symbol b, then $Q(\sigma', \kappa')$ holds with*

$$\begin{aligned} \kappa' &= \mathcal{O}[\![I, E]\!]\sigma(h, b)\kappa \\ \sigma' &= \sigma \circ [E'/I], \\ E' &= \begin{cases} \mathsf{loop}_h(E, I), \text{ if } b = \mathsf{head}, \\ \mathsf{loop}_h(I, E), \text{ if } b = \mathsf{body} \end{cases} \end{aligned}$$

where σ' and κ' take into account the new information introduced by the call to $\mathcal{O}[\![]\!]$. The addition to κ of the code for "$I := E$" in the structural aspect (h, b) of Loop h requires, to maintain consistency, that a new binding be added to σ. This new binding maps the defined variable I to an SSA expression that takes into account where the Imp code has been included, i.e., either in the header or the body of the loop located at Dewey number h. This is what the SSA loop_h expressions are precisely used for. Note that we extend the definition of σ to arbitrary expressions by straightforward structural induction.

As can be seen by the very definition of the Consistency Property, any loop environment with no environment aspects in its domain is consistent with any SSA definition: all code required to compile an SSA binding (I, E) will be regenerated from scratch if need be. The following lemma is thus obvious:

LEMMA 3. *For any SSA code σ, $Q(\sigma, \bot)$.*

We are now equipped to express our final theorem. To generate appropriate Imp code from an SSA binding function, we need to specify the variable I the value of which we are interested in. For the middle end of a compiler that uses SSA as its internal intermediary representation of imperative programs, the codes for all the live variables at the end of the Imp program from which the SSA code is derived need to be concatenated. We use $\mathcal{O}[\![]\!]$ to collect in the head aspect of a "fake" top-level Loop 0 the corresponding Imp code. Since the loop Dewey number 0 is never used in user code, we simply have:

THEOREM 4. *Given $\sigma \in \Sigma$ and an identifier I in the domain of σ. If I_0 is a fresh variable, then the code to compute the value of I is $\kappa(0, \mathsf{head})$, where $\kappa = \mathcal{O}[\![I_0, I]\!]\sigma(0, \mathsf{head})\bot$.*

PROOF. Trivial using Lemma 3 and Theorem 3. □

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 interesting practical and theoretical insights on the computational power of SSA.

7.1 Program Slicing

Program slicing is an optimization technique that extracts, for a given program source, the subset of its instructions that are required to compute a particular facet of the program under analysis, e.g., the value of one of its variables at a given program point (see [27] for a

quick overview of this field). There are numerous applications for program slicing, ranging from debugging to program optimizations to parallelization.

Most approaches to compute a given slice of a program build upon the usually graph-based data structure used to represent the program control-flow, e.g. the Program Dependence Graph [8], and perform a backward analysis of its edges to gather all the instructions required to compute a given variable.

The dual translation processes from and to SSA for imperative programs presented above offer a different means to reach the same goal, without using graph algorithms. Indeed, our SSA-to-Imp specification is implicitly based on the concept of slices, since $\mathcal{O}[\![I, I]\!]\sigma a\kappa$ installs, at Aspect a of κ, a slice for Variable I extracted from the imperative program S if $\sigma = \mathcal{C}[\![S]\!]1\bot$ and I is not already present in κ. Beside its simplicity, one added advantage of our approach is that it provides an immediate inductive correctness proof for program slicing.

Somewhat informally, one could say that SSA "slices" Imp. In the conversion phase from Imp to SSA, the control-flow information contained in the imperative language constructs, such as the sequence of statements or loop nesting, disappears. This information is consumed by the compiler when generating SSA code: as Figure 6 shows, the Dewey h information is not used in the SSA semantic function $\mathcal{E}[\![]\!]$. Conversely, in the conversion to Imp, the Dewey h information is synthesized from the minimal set of dependence relations implicitly contained in the SSA form.

Notice that the compiler to Imp is free to implement different run-time strategies for generating the imperative language information for sequences; the duplication of computations introduced by our $\mathcal{O}[\![]\!]$ specification directly leads to code parallelization, in fact one of the many uses of program slicing.

7.2 The Essence of SSA

All the existing definitions of the SSA form in the literature are influenced by the early papers [11, 12] 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, our paper is the first to give a complete, independent definition of the SSA form, promoting the SSA to the rank of a full-fledged language. An important aspect of SSA is exposed this way: 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 specifying and implementing program optimizations.

Looking now at the conversion process, the mathematical wording of the SSA Conversion Consistency Property (1) underlines one of its key aspects. While p only occurs on the left hand side of the consistency equality, the syntactic location h and the iteration space function k are uncoupled in the right-hand side expression. Thus, via the SSA conversion process, the standard semantics for expression gets staged, informally getting "curryied" from *Expr* $\rightarrow (N^* \times K) \rightarrow T \rightarrow V$ to *Expr* $\rightarrow N^* \rightarrow K \rightarrow T \rightarrow V$; this is also visible on Figure 6, where the pair (h, k) is used on the left arrow, while h and k occur separately on the top and right arrows. This perspective change is rather profound, since it uncouples syntactic sequencing from run time iteration space sequencing.

7.3 Recursive Partial Functions Theory

There exists a formal computing model that is particularly well suited to describing iteration behaviors, namely Kleene's theory of partial recursive functions [34]. In fact, the version of SSA we introduce in this paper appears to be a syntactic variant of such a formalism. We provide below a rewriting $\mathcal{K}[\![]\!]$ of SSA bindings to recursive function definitions.

First, to each SSA identifier I, we associate a function $I(x)$, and translate any SSA expression involving neither loop nor close nodes[6] as function calls:

$$\begin{aligned}\mathcal{K}[\![N]\!]x &= N\\ \mathcal{K}[\![I]\!]x &= I(x)\\ \mathcal{K}[\![E_0 \oplus E_1]\!]x &= \oplus(\mathcal{E}[\![E_0]\!]x, \mathcal{E}[\![E_1]\!]x)\end{aligned}$$

where x is an m-uple $(x_1, \ldots, x_m)$ of syntactic integer variables; m is the number of different Dewey numbers that occur, in loop_h and close_h expressions, in σ. To each h is allocated a slot in x; with a slight abuse of notation, we use h to also denote the index in x of this slot.

Then, to collect partial recursive function definitions corresponding to an SSA program σ, we simply gather all the definitions for each binding, $\bigcup_{I \in Dom\ \sigma} \mathcal{K}[\![I, \sigma I]\!]x$, using $x_{p,q}$ as a shorthand for the $x_p, x_{p+1}, \ldots, x_{q-1}, x_q$ tuple:

$$\begin{aligned}&\mathcal{K}[\![I, \mathsf{loop}_h(E_0, E_1)]\!]x =\\ &\quad \{I(x_{1,h-1}, 0, x_{h+1,m}) = \mathcal{K}[\![E_0]\!](x_{1,h-1}, 0, x_{h+1,m}),\\ &\qquad I(x_{1,h-1}, z+1, x_{h+1,m}) = \mathcal{K}[\![E_1]\!](x_{1,h-1}, z, x_{h+1,m})\}\\ &\mathcal{K}[\![I, \mathsf{close}_h(E_0, E_1)]\!]x =\\ &\quad \{\mathtt{min}_I(x_{1,h-1}, x_{h+1,m}) =\\ &\qquad\qquad (\mu y.\mathcal{K}[\![E_0]\!](x_{1,h-1}, y, x_{h+1,m}) = 0),\\ &\qquad I(x) = \mathcal{K}[\![E_1]\!](x_{1,h-1}, \mathtt{min}_I(x_{1,h-1}, x_{h+1,m}), x_{h+1,m})\}\\ &\mathcal{K}[\![I, E]\!]x = \{I(x) = \mathcal{K}[\![E]\!]x\}\end{aligned}$$

where μ is Kleene's minimization operator. We also assumed that boolean values are coded as integers (*false* is 0). Informally, for loop expressions, we simply rewrite the two cases corresponding to their standard semantics. For close 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.

As an example of this transformation to partial recursive functions, we provide in Figure 8 the translation of our running example into partial recursive functions. For increased readability, we renamed variables to use shorter indices.

$$\begin{aligned}&\mathtt{I}_1(x_1) = 7\\ &\mathtt{J}_1(x_1) = 0\\ &\mathtt{J}_2(0) = \mathtt{J}_1(0)\\ &\mathtt{J}_2(z+1) = \mathtt{J}_3(z)\\ &\mathtt{J}_3(x_1) = +(\mathtt{J}_2(x_1), \mathtt{I}_1(x_1))\\ &\mathtt{min}_{\mathtt{J}_4}() = (\mu y. < (\mathtt{J}_2(y), 10) = 0)\\ &\mathtt{J}_4(x_1) = \mathtt{J}_2(\mathtt{min}_{\mathtt{J}_4}())\end{aligned}$$

Figure 8. Partial recursive functions example.

Our conversion process from Imp to SSA can thus be seen as a way of converting any RAM program [21] to a set of Kleene's partial recursive functions. This and the existence of the dual SSA-to-Imp translation provide a new proof of Turing's Equivalence Theorem between these two computational models, previously typically proven using simulation [21].

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

8. Future Work

Our first goal with this paper is to provide a foundation for specifying the denotational semantics of SSA. This has important future implications since this formalization of the SSA language is the missing stone needed to see other formal frameworks for program analysis, such as abstract interpretation [9, 10], extended to SSA. It will be interesting to see whether our results lead to new insights for program analysis. For instance, regarding abstract interpretation, one intriguing issue is complexity, since most uses of abstract interpretation are based on classical iterative data flow techniques applied to control-flow graphs which introduce an overhead compared to static analysis algorithms working directly on the SSA form [11]. Our approach may be a venue for improvements in this direction.

The second intent of this paper is to investigate the relationship between SSA and the imperative programming paradigm. The two non-standard semantics, namely $\mathcal{C}[\![]$ and $\mathcal{O}[\![]$, that translate programs from one style to the other focus on semantic equivalence and not efficiency, as is evident from Figure 7. In addition to obvious optimizations such as increasing temporary variable reuse or performing constant folding and partial evaluation, our approach could benefit from code duplication removal. This is a serious issue mostly on uniprocessors, since the lack of code sharing can on the contrary be a blessing when dealing with parallel architectures. In fact, as mentioned above, our approach offers all the advantages of program slicing, which is conducive to a high degree of parallelism.

One limitation of our results is that they mainly deal with control issues. On the one hand, this is what SSA is mostly about. But, on the other hand, beyond simple identifiers and values, one would also need to be able to handle more abstract data types such as arrays or objects to tackle full-fledged languages. Even though this would be a serious endeavor, we see this as dual issues to the ones we address in this paper and do not expect them to significantly impact our results, at least as long as such aggregate data structures are considered atomically (see for instance [16] for a more sophisticated approach that strives to detect data dependencies within arrays).

Another restriction imposed by our framework is that, by focusing on only structured abstract syntax trees, unstructured or even irreducible control-flow graphs need to be handled by framing them into such a representation. If this may seem a moot point given the structured design of current programming languages, unstructured control-flow graphs may in fact be more prevalent nowadays than before, given the generalization of exception mechanisms to deal with special cases in otherwise structured code; exceptions do, indeed, destroy the structure of the control flow wherever they are raised. To manage this issue, techniques such as code duplication [1] or control-flow restructuring [15, 2] can be used to recover the program structure required by our approach, although extensions to our SSA language that would deal with this problem in a more direct way may exist.

9. Conclusion

We presented denotational specifications for both the semantics of SSA and its conversion processes to and from a core imperative programming language. Our main theorems show that this semantics is preserved after the transformation of imperative programs to their SSA intermediate form and back. As by-products of our approach, we offer (1) a new way to perform program slicing and (2) another reduction proof for the RAM computational model to Kleene's partial recursive functions theory. All our specifications have been prototyped using GNU Common Lisp.

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. Since our results ensure the correctness of the translation process of all imperative programs to SSA and back, they pave the way to additional research from the programming language community, for instance for static analysis, optimization or parallelization purposes, which would directly target SSA instead of specific source languages. Using SSA as the language of interest for code manipulation and optimization would ensure the portability of the resulting algorithms (see [3] 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, Fortran, C++, Java or Ada via GCC) or functional ones (such as Erlang via HiPE [28]).

Acknowledgments

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

References

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

[2] Z. Ammarguellat. A Control-Flow Normalization Algorithm and its Complexity. *IEEE Transactions on Software Engineering (TOSE)*, 18(3):237–251, 1992.

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

[4] A. W. Appel. SSA is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[5] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 257–271, 1990.

[6] G. Bilardi and K. Pingali. Algorithms for Computing the Static Single Assignment Form. *Journal of the ACM (JACM)*, 50(3):375–425, 2003.

[7] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In IEEE, editor, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 245–255, october 1999.

[8] R. Cartwright and M. Felleisen. The Semantics of Program Dependence. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–27, 1989.

[9] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[10] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 25–35, 1989.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and

the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[13] V. Donzeau-Gouge. Utilisation de la sémantique dénotationnelle pour l'étude d'interprétations non-standard. Technical Report R 273, INRIA, Le Chesnay, France, 1978.

[14] Z. Dvorak. [lno] Enable unrolling/peeling/unswitching of arbitrary loops. GCC Patch Mailing List: `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg02212.html`, march 2004.

[15] A. M. Erosa and L. J. Hendren. Taming Control Flow: A Structured Approach to Eliminating GOTO Statements. In IEEE, editor, *Proceedings of the International Conference on Computer Languages (ICCL)*, pages 229–240, may 1994.

[16] P. Feautrier. Dataflow Analysis of Scalar and Array References. *International Journal of Parallel Programming*, 20(1):23–53, february 1991.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

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

[19] S. Glesner. An ASM Semantics for SSA Intermediate Representations. In *Proceedings of the International Workshop on Abstract State Machines (ASM)*, volume 3052 of *Lecture Notes in Computer Science*. Springer Verlag, may 2004.

[20] J. Guy L Steele. *Common LISP: The Language (Second Edition)*. Digital Press, 1990.

[21] N. D. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.

[22] P. Jouvelot. In and Out of SSA Compilers in GNU Common Lisp. `http://www.cri.ensmp.fr/people/pj/ssa.html`.

[23] P. Jouvelot. Semantic Parallelization: a Practical Exercise in Abstract Interpretation. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 39–48, 1987.

[24] R. A. Kelsey. A Correspondence Between Continuation Passing Style and Static Single Assignment Form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

[25] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):627–676, 1999.

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In IEEE, editor, *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, march 2004.

[27] A. D. Lucia. Program Slicing: Methods and Applications. In IEEE, editor, *First International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 142–149, nov 2001.

[28] D. Luna, M. Pettersson, and K. Sagonas. Efficiently Compiling a Functional Language on AMD64: the HiPE Experience. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 176–186, 2005.

[29] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[30] S. Pop, A. Cohen, and G.-A. Silber. Induction Variable Analysis with Delayed Abstractions. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, volume 3793 of *Lecture Notes in Computer Science*, pages 218–232. Springer Verlag, november 2005.

[31] S. Pop, P. Jouvelot, and G.-A. Silber. In and Out of SSA: a Denotational Specification. Technical Report E-285, CRI/ENSMP, 2007.

[32] L. Presser. Structured Languages. *SIGPLAN Notices*, 10(7):22–24, 1975.

[33] D. Schouten, X. Tian, A. Bik, and M. Girkar. Inside the Intel Compiler. *Linux Journal*, 106, 2003.

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

[35] P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. In ACM, editor, *Proceedings of the International Conference on Supercomputing (ICS)*, pages 414–423, 1995.

[36] M. N. Wegman and F. K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[37] B. Yakobowski. Étude sémantique d'un langage intermédiaire de type Static Single Assignment. Master's thesis, INRIA Rocquencourt, september 2004.

[38] F. K. Zadeck. Loop Closed SSA Form. Personal communication.

[39] F. K. Zadeck. Static Single Assignment Form, GCC and GNU Toolchain Developers' Summit. `http://naturalbridge.com/GCC2004Summit.pdf`, 2004.

Appendix

PROOF OF CONSISTENCY OF STATEMENT CONVERSION THEOREM. By induction on the structure of *Stmt*, assuming $P(\theta, t, p)$:

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

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

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

  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, $P(\theta', t', (h+, k'))$ holds.

- for the sequence $[\![S_0; S_1]\!]$:

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

  By induction, using the result of the theorem on S_0, with $\theta_1 = \mathcal{C}[\![S_0]\!]h.1\theta$, and $(k_1, t_1) = \mathcal{I}[\![S_0]\!]h.1(k, t)$, the property $P(\theta_1, t_1, (h.1+, k_1))$ holds.

Since $h.1+ = h.2$, by induction, using the result of the theorem on S_1, with $\theta_2 = \mathcal{C}[\![S_1]\!]h.2\theta_1$, and $(k_2, t_2) = \mathcal{I}[\![S_1]\!]h.2(k_1, t_1)$, the property $P(\theta_2, t_2, (h+, k_2))$ holds.

So, the property $P(\theta_2, t_2, (h+, k_2))$ holds, since $h.2+ = h+$.

- for the loop $[\![\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{end}]\!]$:

The recursive semantics for while loops suggests to use fixed point induction ([34], 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 4. *With $t = t_0$, $P_0 = P(\theta_{12}, t_0, (h.1, k[0/h]))$ holds.*

This lemma states that if 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. $\forall I \in Dom\ t$:

$$
\begin{array}{ll}
\mathcal{I}[\![I]\!](h.1, k[0/h])t & \\
= R_{<(h.1,k[0/h])}(tI) & (\mathcal{I}[\![]\!]) \\
= R_{<p}(tI) & (t) \\
= \mathcal{I}[\![I]\!]pt & (\mathcal{I}[\![]\!]) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!](\mathcal{R}\sigma)k & (P(\theta, t, h, k)) \\
= \mathcal{E}[\![R_{<h}(\mu I)]\!](\mathcal{R}\sigma)k & (\mathcal{C}[\![]\!]) \\
= \mathcal{E}[\![R_{<h}(\mu I)]\!](\mathcal{R}\sigma)k[0/h] & \text{(first iteration)} \\
= \mathcal{E}[\![R_{<h}(\mu I)]\!](\mathcal{R}\sigma_0)k[0/h] & \text{(extension to } \sigma_0) \\
= \mathcal{E}[\![\mathsf{loop}_h(R_{<h}(\mu I), \bot)]\!](\mathcal{R}\sigma_0)k[0/h] & (\mathsf{loop}_h) \\
= \mathcal{E}[\![\sigma_0 I_{h.0}]\!](\mathcal{R}\sigma_0)k[0/h] & (\sigma_0) \\
= \mathcal{E}[\![I_{h.0}]\!](\mathcal{R}\sigma_0)k[0/h] & (\mathcal{E}[\![]\!]) \\
= \mathcal{E}[\![\mu_0 Ih.0]\!](\mathcal{R}\sigma_0)k[0/h] & (\mu_0) \\
= \mathcal{E}[\![R_{<h.1}(\mu_0 I)]\!](\mathcal{R}\sigma_0)k[0/h] & (R_<) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.1\mu_0]\!](\mathcal{R}\sigma_0)k[0/h] & (\mathcal{C}[\![]\!])
\end{array}
$$

So, $P(\theta_0, t, (h.1, k[0/h]))$ holds. The extension of θ_0 to θ_{12} concludes the proof of Lemma 4. □

LEMMA 5. *Let $(k_x, t_x) = \mathcal{I}[\![S]\!]h.1(k_{x-1}[x - 1/h], t_{x-1})$. Given $P_{x-1} = P(\theta_{12}, t_{x-1}, (h.1, k[x - 1/h]))$ for some $x \geq 1$, then $P_x = P(\theta_{12}, t_x, (h.1, k_x[x/h]))$ holds.*

This second lemma ensures that if 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 $P^0_{x-1} = P(\theta_{12}, t_x, (h.2, k[x - 1/h]))$ holds, since $h.1+ = h.2$, and $\theta_{12} = \mathcal{C}[\![S]\!]h.1\theta_{12}$, as $\mathcal{C}[\![]\!]$ is idempotent. We thus only need now to "go around" to the top of the loop:

$$
\begin{array}{ll}
\mathcal{I}[\![I]\!](h.1, k[x/h])t_x & \\
= R_{<(h.1,k[x/h])}(t_x I) & (\mathcal{I}[\![]\!]) \\
= R_{<(h.2,k[x-1/h])}(t_x I) & (R_<) \\
= \mathcal{I}[\![I]\!](h.2, k[x - 1/h])t_x & (\mathcal{I}[\![]\!]) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.2\mu_1]\!](\mathcal{R}\sigma_2)k[x - 1/h] & (P^0_{x-1}) \\
= \mathcal{E}[\![R_{<h.2}(\mu_1 I)]\!](\mathcal{R}\sigma_2)k[x - 1/h] & (\mathcal{C}[\![]\!]) \\
= \mathcal{E}[\![\mathsf{loop}_h(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))]\!](\mathcal{R}\sigma_2)k[x/h] & (\mathsf{loop}_h) \\
= \mathcal{E}[\![\sigma_2 I_{h.0}]\!](\mathcal{R}\sigma_2)k[x/h] & (\sigma_2) \\
= \mathcal{E}[\![I_{h.0}]\!](\mathcal{R}\sigma_2)k[x/h] & (\mathcal{E}[\![]\!]) \\
= \mathcal{E}[\![\mu_1 Ih.0]\!](\mathcal{R}\sigma_2)k[x/h] & (\mu_1) \\
= \mathcal{E}[\![R_{<h.1}(\mu_1 I)]\!](\mathcal{R}\sigma_2)k[x/h] & (R_<) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.1\mu_1]\!](\mathcal{R}\sigma_2)k[x/h] & (\mathcal{C}[\![]\!])
\end{array}
$$

This concludes the proof of Lemma 5. □

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 $\neg\mathcal{I}[\![E]\!](h.1, k[0/h])t$. Based on Lemma 4, we can show that $P(\theta', t', (h+, k'))$ holds, as $\theta' = \theta_2$ that extends θ_{12}, $t = t'$ as defined by the exit of the while in $\mathcal{I}[\![]\!]$, and $k[0/h] = k'$:

$$
\begin{array}{ll}
\mathcal{I}[\![I]\!](p+)t & \\
= R_{<p+}(tI) & (\mathcal{I}[\![]\!]) \\
= R_{<p}(tI) & (t) \\
= \mathcal{I}[\![I]\!]pt & (\mathcal{I}[\![]\!]) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!](\mathcal{R}\sigma)k & (P) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!](\mathcal{R}\sigma)k' & (k[0/h] = k') \\
= \mathcal{E}[\![R_{<h}(\mu I)]\!](\mathcal{R}\sigma)k' & (\mathcal{C}[\![]\!]) \\
= \mathcal{E}[\![R_{<h}(\mu I)]\!](\mathcal{R}\sigma_2)k' & \text{(extension } \sigma_2) \\
= \mathcal{E}[\![\mathsf{loop}_h(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))]\!](\mathcal{R}\sigma_2)k' & (\mathsf{loop}_h) \\
= \mathcal{E}[\![\sigma_2 I_{h.0}]\!](\mathcal{R}\sigma_2)k' & (\sigma_2) \\
= \mathcal{E}[\![I_{h.0}]\!](\mathcal{R}\sigma_2)k' & (\mathcal{R}) \\
= \mathcal{E}[\![\mathsf{close}_h(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})]\!](\mathcal{R}\sigma_2)k' & (\mathsf{close}_h) \\
= \mathcal{E}[\![\sigma_2 I_{h.2}]\!](\mathcal{R}\sigma_2)k' & (\sigma_2) \\
= \mathcal{E}[\![I_{h.2}]\!](\mathcal{R}\sigma_2)k' & (\mathcal{R}) \\
= \mathcal{E}[\![\mu_2 Ih.2]\!](\mathcal{R}\sigma_2)k' & (\mu_2) \\
= \mathcal{E}[\![R_{<h+}(\mu_2 I)]\!](\mathcal{R}\sigma_2)k' & (R_<) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h+\mu_2]\!](\mathcal{R}\sigma_2)k' & (\mathcal{C}[\![]\!])
\end{array}
$$

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:

$$
\begin{array}{ll}
\omega = \min\{x \mid \neg\mathcal{I}[\![E]\!](h.1, k[x/h])t_x\} & \\
\quad = \min\{x \mid \neg\mathcal{E}[\![\mathcal{C}[\![E]\!]h.1\mu_1]\!](\mathcal{R}\sigma_2)k[x/h]\} & \text{(Lemma 1)}
\end{array}
$$

By Lemmas 4 and 5, using induction on S, we know that $P^0_\omega = P(\theta_{12}, t_\omega, (h.2, k[\omega - 1/h]))$ holds. We prove be-

low that $P(\theta', t_\omega, (h+, k'))$ also holds (as a shorthand, we note $k^n = k[n/h]$):

$$
\begin{array}{ll}
\mathcal{I}[\![I]\!](p+)t_\omega & \\
= R_{<p+}(t_\omega I) & (\mathcal{I}[\![]\!]) \\
= R_{<(h.2,k^{\omega-1})}(t_\omega I) & (R_<) \\
= \mathcal{I}[\![I]\!](h.2, k^{\omega-1})t_\omega & (\mathcal{I}[\![]\!]) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.2\mu_1]\!](\mathcal{R}\sigma_2)k^{\omega-1} & (P^0_\omega) \\
= \mathcal{E}[\![R_{<h.2}(\mu_1 I)]\!](\mathcal{R}\sigma_2)k^{\omega-1} & (\mathcal{C}[\![]\!]) \\
= \mathcal{E}[\![\mathsf{loop}_h(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))]\!](\mathcal{R}\sigma_2)k^\omega & (\mathsf{loop}_h) \\
= \mathcal{E}[\![\sigma_2 I_{h.0}]\!](\mathcal{R}\sigma_2)k^\omega & (\sigma_2) \\
= \mathcal{E}[\![I_{h.0}]\!](\mathcal{R}\sigma_2)k^\omega & (\mathcal{R}) \\
= \mathcal{E}[\![\mathsf{close}_h(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})]\!](\mathcal{R}\sigma_2)k & (\mathsf{close}_h) \\
= \mathcal{E}[\![\sigma_2 I_{h.2}]\!](\mathcal{R}\sigma_2)k & (\sigma_2) \\
= \mathcal{E}[\![I_{h.2}]\!](\mathcal{R}\sigma_2)k & (\mathcal{R}) \\
= \mathcal{E}[\![\mu_2 I h.2]\!](\mathcal{R}\sigma_2)k & (\mu_2) \\
= \mathcal{E}[\![R_{<h+}(\mu_2 I)]\!](\mathcal{R}\sigma_2)k & (R_<) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h+\mu_2]\!](\mathcal{R}\sigma_2)k & (\mathcal{C}[\![]\!])
\end{array}
$$

Finally, using Kleene's Fixed Point Theorem [34], we can relate the least fixed point $\mathrm{fix}(W_h)$ used to define the standard semantics of while loops and the successive iterations $W^i_h(\bot)$ of the loop body:

$$
\begin{array}{rcl}
t' & = & \mathrm{fix}(W_h)(k[0/h], t) \\
& = & \lim\limits_{i\to\infty} W^i_h(\bot)(k[0/h], t) \\
& = & W^\omega_h(\bot)(k[0/h], t) \\
& = & t_\omega
\end{array}
$$

and so $P(\theta', t', (h+, k'))$ holds.

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

$$
\begin{array}{ll}
\mathcal{I}[\![I]\!](p+)\bot & \\
= \bot & (\mathcal{I}[\![]\!]) \\
= \mathcal{E}[\![\mathsf{close}_h(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})]\!](\mathcal{R}\sigma_2)k & (\min\emptyset = \bot) \\
= \mathcal{E}[\![\sigma_2 I_{h.2}]\!](\mathcal{R}\sigma_2)k & (\sigma_2) \\
= \mathcal{E}[\![I_{h.2}]\!](\mathcal{R}\sigma_2)k & (\mathcal{R}) \\
= \mathcal{E}[\![\mu_2 I h.2]\!](\mathcal{R}\sigma_2)k & (\mu_2) \\
= \mathcal{E}[\![R_{<h+}(\mu_2 I)]\!](\mathcal{R}\sigma_2)k & (R_<) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h+\mu_2]\!](\mathcal{R}\sigma_2)k & (\mathcal{C}[\![]\!])
\end{array}
$$

So, $P(\theta', t', (h+, k'))$ holds.

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

PROOF OF OUT OF SSA EXPRESSION CONSISTENCY.

We use the following lemmas in the proof, which deal with the impact of the $\mathsf{up}[\![]\!]$ helper function on the Consistency Property .

LEMMA 6 (Consistency of $\mathsf{up}[\![]\!]$ for Aspect $a = (h, \mathsf{head})$). *Assume an execution point $p = (h, k)$, a state t, a variable I and a "simple" expression E (a constant, variable or $\oplus$ of simple expressions). If $Q(\sigma, \kappa)$, $I \notin \mathsf{dom}_{\mathsf{env}}(\kappa)$, I doesn't occur in E and all variables of E are unbound in*[7] *$Un = \{(h, \mathsf{body}), (h, \mathsf{iter})\}$, then $Q(\sigma', \kappa')$ holds with:*

$$
\begin{array}{l}
\sigma' = \sigma \circ [\mathsf{loop}_h(E, I)/I] \\
\kappa' = \mathsf{up}[\![I{:=}E]\!](h, \mathsf{head})\kappa
\end{array}
$$

Proof: The Out of SSA Consistency Property is obvious for all variables, but I. Let

$$
\begin{array}{l}
(k_0, t_0) = \mathcal{I}[\![\kappa(h, \mathsf{head}); (\kappa(h, \mathsf{body}); \kappa(h, \mathsf{iter}))^{kh}]\!]h(k, t) \\
(k', t') = \mathcal{I}[\![\kappa'(h, \mathsf{head}); (\kappa'(h, \mathsf{body}); \kappa'(h, \mathsf{iter}))^{kh}]\!]h(k, t)
\end{array}
$$

Then,

$$
\begin{array}{ll}
\mathcal{E}[\![I]\!](\mathcal{R}\sigma')k & \\
= (\mathcal{R}\sigma')Ik & (\mathcal{E}[\![]\!]) \\
= (\mathcal{R}(I, \mathsf{loop}_h(E, I))(\mathcal{R}\sigma'))Ik & (\sigma') \\
= ((\mathcal{R}\sigma')[\lambda x.\mathcal{E}[\![\mathsf{loop}_h(E, I)]\!](\mathcal{R}\sigma')x/I])Ik & (\mathcal{R}) \\
= \mathcal{E}[\![\mathsf{loop}_h(E, I)]\!](\mathcal{R}\sigma')k & (\text{apply to } I \text{ and } k) \\
= (1) & \\
= \mathcal{E}[\![E]\!](\mathcal{R}\sigma)k & (I \text{ unused in } E) \\
= \mathcal{I}[\![E]\!](h+, k_0)t_0 & (2) \\
= \mathcal{I}[\![I]\!](h+, k')t' & (3)
\end{array}
$$

(1) = $\mathcal{E}[\![E]\!](\mathcal{R}\sigma')k$ for $kh = 0$, by very definition of $\mathcal{E}[\![]\!]$. Since $\sigma' = \sigma \circ [\mathsf{loop}_h(E, I)/I]$, for $kh > 0$ one gets the same value by induction on kh, using the definition of $\mathcal{E}[\![]\!]$ ($\mathcal{E}[\![I]\!](\mathcal{R}\sigma')k_{h-}$), i.e. the fact that the value of I is defined when $kh = 0$ and not modified in the subsequent iterations.

(2) by definition of $\mathcal{I}[\![]\!]$, structural induction on E using the lemma hypothesis and the fact that the syntactic translation of simple expressions E from SSA to Imp is the identity.

(3) since $\kappa'(h, \mathsf{head}) = \kappa(h, \mathsf{head}); I := E$ and the statements in $\kappa'(h, \mathsf{body})$ and in $\kappa'(h, \mathsf{iter})$ do not contain assignments to I or variables of E. □

LEMMA 7 (Consistency of $\mathsf{up}[\![]\!]$ for Aspect $a = (h, \mathsf{body})$). *Similar to Lemma 6, with*

$$
\begin{array}{l}
Un = \{(h, \mathsf{iter})\} \\
\sigma' = \sigma \circ [\mathsf{loop}_h(I, E)/I] \\
\kappa' = \mathsf{up}[\![I{:=}E]\!](h, \mathsf{body})\kappa
\end{array}
$$

Proof similar to the one for Lemma 6, except that:

(1) = $\mathcal{E}[\![I]\!](\mathcal{R}\sigma')k$ for $kh = 0$, by very definition of $\mathcal{E}[\![]\!]$; the value of I is $\bot$ (undefined) in both $\mathcal{E}[\![]\!]$ and $\mathcal{I}[\![]\!]$ (since the loop body is not executed). For $kh > 0$, (1) = $\mathcal{E}[\![E]\!](\mathcal{R}\sigma')k_{h-}$, using the definition of $\mathcal{E}[\![]\!]$. Since $\sigma' = \sigma \circ [\mathsf{loop}_h(I, E)/I]$, the value of I is, at every iteration $kh > 0$, the one of E; only the last value matters (since $I \notin \mathsf{dom}_{\mathsf{env}}(\kappa)$, the code in $\kappa'(h, \mathsf{iter})$ does not contain assignments to I).

(3) since $\kappa'(h, \mathsf{body}) = \kappa(h, \mathsf{body}); I := E$ and the code in $\kappa'(h, \mathsf{iter})$ does not contain assignments to I or variables of E.□

[7] A variable I is *unbound in Un* iff for all $(h, b) \in Un$, $(I, b) \notin \kappa(h, \mathsf{env})$.

LEMMA 8 (Consistency of $\mathsf{up}[\![\,]\!]$ for Aspect $a = (h, \mathsf{iter})$). *Similar to Lemma 7, with no constraints on I and*

$$Un = \emptyset$$
$$\kappa' = \mathsf{up}[\![I{:=}E]\!](h, \mathsf{iter})\kappa$$

Proof similar to the one for Lemma 7, with

(3) $\kappa'(h, \mathsf{iter}) = \kappa(h, \mathsf{iter}); I := E$ and the assignment to I is the last in the loop body. □

Assuming $Q(\sigma, \kappa)$, the proof of the main theorem looks at $\mathcal{O}[\![I, E]\!]\sigma a\kappa$, where $I \notin \mathsf{dom}_{\mathsf{env}}(\kappa)$. It uses a double induction on (1) the number of identifiers in the domain of σ present in $\mathsf{dom}_{\mathsf{env}}(\kappa)$ and (2) the structure of the SSA expression E:

- for N, $Q(\sigma', \kappa')$ holds, by Lemmas 6 and 7.
- for $I', I' \in \mathsf{dom}_{\mathsf{env}}(\kappa)$, $Q(\sigma', \kappa')$ holds, by Lemmas 6 and 7. Indeed, analyzing the defining cases for $\mathcal{O}[\![]\!]$, all calls $\mathsf{up}[\![I, I := I']\!]\sigma a\kappa$ where $I' \in \mathsf{dom}_{\mathsf{env}}(\kappa)$ are such that κ only includes a call $\mathcal{O}[\![I', \ldots]\!]\sigma a_0\kappa_0$ where $a = a_0$ or $a = (h, \mathsf{iter})$ and $a_0 = (h, \mathsf{body})$. In both cases, I' is unbound in *Un*.
- for $I', I' \notin \mathsf{dom}_{\mathsf{env}}(\kappa)$, $Q(\sigma', \kappa')$ holds with

$$\kappa_0 = \mathcal{O}[\![I', \sigma I']\!]\sigma a(\mathsf{up}_{\mathsf{env}}[\![I']\!]a\kappa)$$

$$\begin{aligned} & Q(\sigma, \kappa) \\ & \Rightarrow Q(\sigma_0, \kappa_0) \qquad (1) \\ & \Rightarrow Q(\sigma', \kappa') \qquad (2) \end{aligned}$$

(1) by induction on $|Dom\ \sigma - \mathsf{dom}_{\mathsf{env}}(\kappa)|$, with $\sigma_0 = \sigma \circ [\mathsf{loop}_h(\sigma I', I')/I']$. Note that even though $I' \in \mathsf{dom}_{\mathsf{env}}(\mathsf{up}_{\mathsf{env}}[\![I']\!]a\kappa)$, the use of the conclusion of Theorem 3 is valid, since we use for E the expression $\sigma I'$ that defines I'.

(2) by Lemma 6, for $a = (h, \mathsf{head})$, as above. One then gets:

$$\begin{aligned} \sigma' &= (\sigma \circ [\mathsf{loop}_h(\sigma I', I')/I']) \circ [\mathsf{loop}_h(I', I)/I] \\ &= \sigma \circ [\mathsf{loop}_h(\mathsf{loop}_h(\sigma I', I'), I)/I] \\ &= \sigma \circ [\mathsf{loop}_h(\sigma I', I)/I] \\ &= \sigma \circ [\mathsf{loop}_h(I', I)/I] \qquad (\mathcal{E}[\![]\!]) \end{aligned}$$

as requested. The proof for (h, body) uses Lemma 7.

- for $E_0 \oplus E_1$, $Q(\sigma', \kappa')$ holds:

$$\kappa_0 = \mathcal{O}[\![I_0, E_0]\!]\sigma a\kappa$$
$$\kappa_1 = \mathcal{O}[\![I_1, E_1]\!]\sigma a\kappa_0$$
$$\sigma_0 = \sigma \circ [\mathsf{loop}_h(E_0, I_0)/I_0]$$
$$\sigma_1 = \sigma_0 \circ [\mathsf{loop}_h(E_1, I_1)/I_1]$$

$$\begin{aligned} Q(\sigma, \kappa) &\Rightarrow Q(\sigma_0, \kappa_0) \qquad \text{(induction on } E_0\text{)} \\ &\Rightarrow Q(\sigma_1, \kappa_1) \qquad \text{(induction on } E_1\text{)} \\ &\Rightarrow Q(\sigma', \kappa') \qquad \text{(*)} \end{aligned}$$

(*) using, for $a = (h, \mathsf{head})$, Lemma 6 with $E = I_0 \oplus I_1$, yielding $\sigma' = \sigma_1 \circ [\mathsf{loop}_h(I_0 \oplus I_1, I)/I]$. By definition of σ_0 and σ_1, this can be successively rewritten as $\sigma' = \sigma_0 \circ [\mathsf{loop}_h(I_0 \oplus \mathsf{loop}_h(E_1, I_1), I)/I] = \sigma \circ [\mathsf{loop}_h(\mathsf{loop}_h(E_0, I_0) \oplus \mathsf{loop}_h(E_1, I_1), I)/I]$. By distributivity of loop over $\oplus$, one gets $\sigma' = \sigma \circ [\mathsf{loop}_h(\mathsf{loop}_h(E_0 \oplus E_1, I_1 \oplus I_0), I)/I]$, which is $\sigma \circ [\mathsf{loop}_h(E_0 \oplus E_1, I)/I]$, by definition of the semantics of loop; this is the expected formula for σ' required to complete the proof. Similarly, the proof for $a = (h, \mathsf{body})$ uses Lemma 7.

- for $\mathsf{loop}_{h_1}(E_0, E_1)$, $Q(\sigma', \kappa')$ holds:

$$\kappa_0 = \mathcal{O}[\![I, E_0]\!]\sigma(h_1, \mathsf{head})\kappa$$
$$\kappa_1 = \mathcal{O}[\![I_1, E_1]\!]\sigma(h_1, \mathsf{body})\kappa_0$$
$$\sigma_0 = \sigma \circ [\mathsf{loop}_{h_1}(E_0, I)/I]$$
$$\sigma_1 = \sigma_0 \circ [\mathsf{loop}_{h_1}(I_1, E_1)/I_1]$$

$$\begin{aligned} Q(\sigma, \kappa) &\Rightarrow Q(\sigma_0, \kappa_0) \qquad \text{(induction on } E_0\text{)} \\ &\Rightarrow Q(\sigma_1, \kappa_1) \qquad \text{(induction on } E_1\text{)} \\ &\Rightarrow Q(\sigma', \kappa') \qquad \text{(*)} \end{aligned}$$

(*) using Lemma 8 with $E = I_1$, one gets, by definition of σ_1 and σ_0 and semantics of loop:

$$\begin{aligned} \sigma' &= \sigma_1 \circ [\mathsf{loop}_{h_1}(I, I_1)/I] \\ &= \sigma_0 \circ [\mathsf{loop}_{h_1}(I, \mathsf{loop}_{h_1}(I_1, E_1))/I] \\ &= \sigma \circ [\mathsf{loop}_{h_1}(\mathsf{loop}_{h_1}(E_0, I), \mathsf{loop}_{h_1}(I_1, E_1))/I] \\ &= \sigma \circ [\mathsf{loop}_{h_1}(E_0, E_1)/I] \end{aligned}$$

Note that the proof is independent of a.

- for $\mathsf{close}_{h_1}(E_0, E_1)$, $Q(\sigma', \kappa')$ holds:

$$\kappa_0 = \mathcal{O}[\![I_0, \mathsf{loop}_{h_1}(E_0, E_0)]\!]\sigma a\kappa$$
$$\kappa_1 = \mathcal{O}[\![I_1, \mathsf{loop}_{h_1}(E_1, E_1)]\!]\sigma a\kappa_0$$
$$\kappa' = \mathsf{up}[\![I, W; I := I_1]\!]a\kappa_1$$
$$\sigma_0 = \sigma \circ [\mathsf{loop}_h(\mathsf{loop}_{h_1}(E_0, E_0), I_0)/I_0]$$
$$\sigma_1 = \sigma_0 \circ [\mathsf{loop}_h(\mathsf{loop}_{h_1}(E_1, E_1), I_1)/I_1]$$

$$\begin{aligned} Q(\sigma, \kappa) &\Rightarrow Q(\sigma_0, \kappa_0) \qquad \text{(induction on loop } E_0\text{)} \\ &\Rightarrow Q(\sigma_1, \kappa_1) \qquad \text{(induction on loop } E_1\text{)} \\ &\Rightarrow Q(\sigma', \kappa') \qquad \text{(*)} \end{aligned}$$

(*) In the absence of W, for $a = (h, \mathsf{head})$, using Lemma 6, the induction would yield a consistent state with $\sigma' = \sigma \circ [\mathsf{loop}_h(\mathsf{loop}_{h_1}(E_1, E_1), I)/I]$. For $a = (h, \mathsf{body})$, one would get a similar result, with the loop expression and I swapped. For any aspect, I would be bound in σ' to a loop that iterates over E_1; I is always equal to E_1. A similar result exists for I_0 and E_0.

The statement W located at h_W, if its execution terminates in the rolling state (k', t') after starting in (k, t), is, by definition of $\mathcal{I}[\![]\!]$, semantically equivalent to

$$\kappa_1(h_1, \mathsf{head}); (\kappa_1(h_1, \mathsf{body}); \kappa_1(h_1, \mathsf{iter}))^\omega,$$

where

$$\omega = \min\{x \mid \neg\mathcal{I}[\![I_0]\!](h_1.1, k[x/h_1])t_x\},$$

where t_x denotes the state after x iterations of the loop, with $t_0 = t$. Note that, since $Q(\kappa_1, \sigma_1)$, the values of all variables of $\mathsf{dom}_{\mathsf{env}}(\kappa_1)$, after executing the unrolled loop, are independent of h_W and t

By continuity, if W doesn't terminate, then ω is the minimum of the empty set, i.e., $+\infty$, and we can keep the same definition and semantic equivalence.

Thus, from any starting rolling state (k, t), the sequence $W; I := I_1$ located at h imposes that the value $\mathcal{I}[\![I_1]\!]((h.1)+, k')t'$ of I_1, latter assigned to I, is determined in a state (k', t') that is equivalent to the one mentioned in Definition 2 with $kh_1 = \omega$ iterations of Loop W, located at $h.1$. Since the Out of SSA Consistency property using the rolling state $(k[\omega/h_1], t_\omega)$ ensures that, for I_1:

$$\mathcal{I}[\![I_1]\!]((h.1)+, k')t' = \mathcal{E}[\![I_1]\!](\mathcal{R}\sigma_1)k[\omega/h_1]$$

and similarly for I_0:

$$\begin{aligned}\omega &= \min\{x \mid \neg\mathcal{I}[\![I_0]\!]((h.1).1, k[x/h_1])t_x\}\\ &= \min\{x \mid \neg\mathcal{E}[\![I_0]\!](\mathcal{R}\sigma_1)k[x/h_1]\},\end{aligned}$$

the value of I_1, and hence I, is thus, in the presence of W:

$$\mathcal{E}[\![I_1]\!](\mathcal{R}\sigma_1)k[\min\{x \mid \neg\mathcal{E}[\![I_0]\!](\mathcal{R}\sigma_1)k[x/h_1]\}/h_1],$$

i.e., $\mathcal{E}[\![E']\!](\mathcal{R}\sigma_1)k$, with $E' = \mathsf{close}_{h_1}(I_0, I_1)$, which yields, by definition of $\mathcal{E}[\![]\!]$ and substitutions using σ_1:

$$\mathsf{close}_{h_1}(\mathsf{loop}_h(\mathsf{loop}_{h_1}(E_0, E_0), I_0), \mathsf{loop}_h(\mathsf{loop}_{h_1}(E_1, E_1), I_1))$$

For the theorem to be true, we need to have $\sigma' = \sigma \circ [\mathsf{loop}_h(\mathsf{close}_{h_1}(E_0, E_1), I)/I]$. This requires us to show that E' and $\mathsf{loop}_h(\mathsf{close}_{h_1}(E_0, E_1), I)$ are equal.

For $kh \neq 0$, both SSA expressions evaluate to $\perp$. When $kh = 0$, E' is equivalent to $\mathsf{close}_{h_1}(\mathsf{loop}_{h_1}(E_0, E_0), \mathsf{loop}_{h_1}(E_1, E_1))$, while the second is $\mathsf{close}_{h_1}(E_0, E_1)$; they need to be shown equivalent. The semantics of E' is

$$\mathcal{E}[\![\mathsf{loop}_{h_1}(E_1, E_1)]\!]k[\min\{x \mid \neg\mathcal{E}[\![\mathsf{loop}_{h_1}(E_0, E_0)]\!]k[x/h_1]\}/h_1].$$

If the minimum, ω, is 0, then $\mathcal{E}[\![\mathsf{loop}_{h_1}(E_1, E_1)]\!]k[\omega/h_1] = \mathcal{E}[\![E_1]\!]k[\omega/h_1]$. A similar reasoning works for $\mathsf{loop}_{h_1}(E_0, E_0)$. The value of E' is thus the same as the one of $\mathsf{close}_h(E_0, E_1)$.

If ω is not 0, then the semantics of E' is

$$\mathcal{E}[\![E_1]\!]k[\min\{x \mid \neg\mathcal{E}[\![E_0]\!]k[x - 1/h_1]\} - 1/h_1].$$

Defining $y = x - 1$, one can rewrite this as:

$$\mathcal{E}[\![E_1]\!]k[\min\{y + 1 \mid \neg\mathcal{E}[\![E_0]\!]k[y/h_1]\} - 1/h_1],$$

which, by distributing $+1$ over $\min$, is the same as $\mathcal{E}[\![E_1]\!]k[\omega/h_1]$. The value of E' is thus here also the same as the one of $\mathsf{close}_{h_1}(E_0, E_1)$. □