Categorical Imperative Programming

Type theory, refinement, and semantics for SSA

Jad-Elkhaleq Ghalayini

March 2026

UNIVERSITY OF CAMBRIDGE
Department of Computer Science and Technology

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted, or is being concurrently submitted, for any degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Contents

1. Introduction

The fundamental question of this thesis is: *what does a computer program mean?*

We can approach this in two ways: from the bottom-up or the top-down.

The former tends to be more natural for computer scientists.

A computer is a machine for following instructions – a program is a sequence of instructions for it to follow.

In particular, a computer has an *instruction set* – a finite set of *operations* which it can perform, which read from and write to its *state* – that is, the current state of the registers, program counters, memory, interrupts, and all the other bits which make a modern CPU tick.

Which is a *lot*.

This thesis is about the *denotational semantics* of the *static single assignment* – or *SSA* – compiler *intermediate representation* – or *IR*.

I want to start by justifying this choice of topic.

- *Why* do we need a semantics for a *compiler IR*?

  Because high-level languages have too many features to effectively study *or* to lower in one pass.

  Because low-level languages are designed to be *executed* efficiently – making them difficult both to reason about or to lower in one pass.

- *Why* should we use *SSA* as our compiler IR?

  Practically: most modern compilers use it.

  Theoretically: it's right at the intersection of functional and imperative programming – letting us use it as a *thin waist* to develop the *isomorphism* between the two.

- *Why* should we give a *denotational* semantics?

  Because we want to reason about programs *algebraically* – and to do so in a *compositional* manner.

2. Type-Theoretic SSA

> Premature optimization is the root of all evil.
>
> — Donald Knuth

The goal of this chapter is to give a formal presentation of SSA – fully specifying its *syntax*, *type theory*, and *semantics*. Rather than do this all at once, we're instead going to "build up to" SSA by studying a sequence of increasingly complex languages, each of which embeds into the next.

In particular, we'll begin by introducing:

1. The language of simple arithmetic expressions, λ_{arith}, which we'll extend to:
2. The language of *effectful* expressions, λ_{expr}, which we'll restrict to:
3. The language of *operations*, λ_{op} – a single effectful expression which depends on variables and constants. This will allow us to define

4. The language of *basic blocks*, λ_{bb}, – straight-line sequences of operations – which we'll show *equivalent* to the language of effectful expressions with *let-bindings*. By adding conditional branches, we can then recover
5. The language of *acyclic regions*, λ_{areg}, – single-entry, single-exit *acyclic* control-flow graphs of basic blocks – which we'll show *equivalent* to the language of effectful expressions with *case-expressions*

Finally, we'll drop the acyclicity restriction to get to our final type-theoretic presentation of SSA:

6. The language of *regions*, λ_{reg}, – single-entry, multiple-exit control-flow graphs of basic blocks – which we'll show *equivalent* to the language of effectful expressions with *iteration*

Our goal is to give a precise mathematical description of each of these languages – consequently, each language's section will follow a somewhat repetitive pattern, where we specify:

- A *syntax*, which specifies how to tell whether a term is *grammatical* – like "$x * (y + z)$" – or nonsense – like "$C++$".
- A *type system*, which consists of:
  - A collection of ways we can *interpret* a given term (natural number, string, graph, tensor, neural network…) – that is, a set of *types* A, B, C
  - A description of the *contexts* in which we can interpret terms
  - A *typing relation* – which tells us in which contexts a given term can be interpreted in a given way.

    Concretely: we might be able to interpret $x + x$ as an integer in the context $x : \mathbb{Z}$, or as a string in the context $x : \texttt{str}$ – but $x + x$ doesn't make sense for $x : \texttt{POBox}$.

Normally, a *type theory* consists of a type system together with an *equational theory*: a relation which tells us when two terms are "the same" in a particular context – for example, we might have $x + y \equiv y + x$ in the context $x : \mathbb{Z}, y : \mathbb{Z}$, but not in the context $x : \texttt{str}, y : \texttt{str}$.

This can let us reason about programs, but also *optimize* them – for example, we might rewrite

$$(x + 5) + (x + 5) + (x + 5) \qquad \longrightarrow \qquad \mathsf{let}\ y = x + 5; 3 \cdot y$$

Since our main focus will be the study of compiler optimizations, we will hence generalize equationa theories to *refinement theories*: where a refinement theory is a relation which tells us that replacing a term t with another term t' is always sound, i.e., that t' *refines* t – written $t \to t'$.

We can recover an equational theory by saying that t and t' are *equivalent* – written $t \approx t'$ – whenever $t \to t'$ and $t' \to t$. Likewise, every equational theory induces a refinement theory by simply defining $(t \to t') \coloneqq (t \approx t')$.

As a concrete example, let's assume that the result of $x/0$ is *implementation-defined* – and, in particular, may return a different result each time it's evaluated.

Then it's sound to rewrite

$$(x/y) + (x/y) \qquad \longrightarrow \qquad \mathsf{let}\ z = x/y; z + z$$

since, whether or not $y = 0$, every valid result for the original program is also a valid result for the rewritten program.

On the other hand, for $x, y : \mathbb{Z}$, the rewrite

$$\mathsf{let}\ z = x/y; 2 \cdot z \qquad \not\longrightarrow \qquad (x/y) + (x/y)$$

would be unsound, since every valid result for the left-hand side is *even* (since it can be written $2 \cdot z$ for some z) but the right-hand side can be *odd* – for example, if $y = 0$, we could have the reduction sequence

$$(x/y) + (x/y) \longrightarrow 1 + (x/y) \longrightarrow 1 + 0 \longrightarrow 1$$

Just like equations, refinements can be *context-dependent* – for example, depending on our semantics, we may have that

$$\mathsf{let}\ z = x/y; 2 \cdot z \qquad \longrightarrow \qquad (x/y) + (x/y)$$

is a perfectly valid optimization for $x, y : \mathbb{R}$.

At this point, we've got a *theory* for our programming language:

- A *syntax*, which sets the domain of discourse
- A *type theory*, which tells us what rules we can use to reason about sentences in our syntax

What we now need to confirm is whether this theory is a *good* one – that is, whether it captures our intuitions about how programs should behave.

To do this, we need to give our theory a *semantics*. There are two primary approaches we might take:

- We can provide an *operational semantics*: a relation which specifies the behaviour of a program by describing how it would be executed by an *abstract machine*.

  Our semantics is then *sound* if $t \to t'$, implies that valid execution of t' is a valid execution of t.

  Alternatively,

- We can provide a *denotational semantics*: a mapping from well-formed terms in our syntax to mathematical objects.

  Soundness in this case means that every sentence which is true according to our type theory is true when interpreted as a statement about the semantics.

  For example, if we interpret programs like $x/(y + z)$ as functions from *valuations* $\mathsf{Var} \to \mathbb{Z}$ to sets of possible results $\mathcal{P}(\mathbb{Z})$, then we might say our type theory is *sound* if, for all well-formed t, t',

  $$t \to t' \implies [\![t]\!] \subseteq [\![t']\!] \qquad \text{(pointwise)}$$

The standard approach for reasoning about SSA – e.g. in **TODO 1: VELVVM, etc** – is operational. One of the primary contributions of this thesis is to give a *compositional* denotational semantics for SSA – that is, a denotational semantics in which the semantics of a program as a whole is a function of the semantics of its parts. In particular, this will be enabled by *categorical semantics*: where types and contexts are intepreted as *objects* in a *category*.

While the resulting semantics enables compositional reasoning about SSA programs, it is also quite abstract – hence, to validate that it is in fact the "right" semantics, we'll show that:

- It is *complete* with respect to our refinement theory: that is, $t \to t'$ if and only if $[\![t]\!]_{\mathcal{M}} \leq [\![t']\!]_{\mathcal{M}}$ for every model $\mathcal{M}$
- Our refinement theory is *sound* with respect to a reasonable operational semantics – and hence our categorical semantics at least *abstracts* the operational semantics.

Let's get started!

2.1. Iteration Expressions

SSA is a useful IR because it enables complex, whole-procedure transformation of code, but its block-statement-value hierarchy makes it difficult to uniformly formulate the allowed transformations as an (in)equational theory. Instead, therefore, we will start by studying a simple, first-order language of expressions, λ_{iter}, equipped with an *iteration operator* – we will then show that this language is *equivalent* to SSA, and therefore we'll be able to use it as a tool to more effectively give a sound and complete equational, and later refinement, theory for SSA.

2.1.0.1. Syntax

λ_{iter} is a standard first-order expression language with branching and iteration: a functional analogue of While. Its grammar is parametrized by a set of *base types* $X \in \mathcal{X}$ and a set of *instructions* $f \in \mathcal{I}$. To model multiple arguments and control flow, our grammar of types A, B, C includes all *tensor products* $A \otimes B$ and *coproducts* $A + B$ generated by base types X, along with a *unit type* $\mathbf{1}$ and an *empty type* $\mathbf{0}$. Mirroring this, expressions a, b, c consist of

- *Variables* x, y, z
- *Applications* $f\ a$, consisting of instructions $f \in \mathcal{I}$ applied to an expression a
- *Let-bindings* $\mathsf{let}\ x = a\ b$ and *destructuring let-bindings* $\mathsf{let}\ (x, y) = a\ b$. We write $a; b$ as syntactic sugar for $\mathsf{let}\ \cdot = a\ b$ (i.e., a let-binding where the bound variable is not used in b).
- *Case-expressions* $\mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\}$, representing branching control-flow
- *Iteration-expressions* $\mathsf{iter}\ a\ \{\iota_r\ x : b\}$, representing a variant of *tail-controlled loop*, which:
  - Evaluates an initial value $a : A$
  - Evaluates the loop body $b : B + A$ with $x = a$; if the loop evaluates to a value of type B, we return it, otherwise, we re-evaluate the loop with x having the new value of type A

We give the full grammar for λ_{iter} types, expressions, and contexts in Figure 1.

$$
\begin{array}{rcl}
A, B, C & ::= & X \mid A \otimes B \mid \mathbf{1} \mid A + B \mid \mathbf{0} \\
a, b, c & ::= & x \mid f\ a \mid \mathsf{let}\ x = a\ b \mid () \mid (a, b) \mid \mathsf{let}\ (x, y) = a\ b \mid \iota_l\ a \mid \iota_r\ b \mid \mathsf{case}\ a\ \{\iota_l\ x : b, \iota_r\ y : c\} \mid \mathsf{abort}\ a \mid \mathsf{iter}\ a\ \{\iota_r\ x : b\} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Figure 1: Syntax for λ_{iter} types, expressions, and contexts.

We can then give typing rules for λ_{iter} as follows: **TODO 2: give raw typing rules**

TODO 3: weakening

TODO 4: substitution

TODO 5:

Bibliography

1. Sample Appendix

SAMPLE TEXT