Compiler Generation for Substructural Operational Semantics

Anand Subramanian

CMU-CS-12-150

December 19, 2012

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, Chair
Iliano Cervesato (Carnegie Mellon Qatar)

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science.*

Copyright © 2012 Anand Subramanian

Keywords: Linear Logic, Operational Semantics, Interpreter, Compiler

Abstract

Semantic specifications of programming languages can be used to assist or automate compiler generation. Semantics-driven compiler generation has already been studied for specification techniques such as denotational semantics, natural semantics, structural operational semantics and action semantics. Compilers generated from such specifications bring some of the performance benefits of traditional compilers to prototype specifications of programming languages while reducing or eliminating the logistical overhead of implementing a separate piece of software and proving its correctness.

This dissertation describes techniques that can be used to synthesize a compiler and virtual machine from a Substructural Operational Semantics (SSOS). Whereas prior work relied on techniques such as partial evaluation and staged computation to derive compilers from interpreters, we use linear logical approximations for synthesis. Our methodology is illustrated using language features from C0, a safe subset of C used to teach imperative programming at Carnegie Mellon University.

Acknowledgments

I owe the successful completion of this dissertation to my advisor, Frank Pfenning. For most of my years as an undergraduate and this past year as a graduate student, Frank has taught courses that I have taken and given me valuable advise that has not only influenced this dissertation, but informed me about logic and computation on the whole. I thank Iliano Cervesato for his valuable and timely feedback which has helped to significantly improve the quality of this document. I also thank Karl Crary, whose tutelage helped me develop proficiency in logic as an undergraduate.

I cannot overstate the importance of the intellectual environment created by members of the CMU POP group. Rob Simmons in particular helped brainstorm and bring out the best in even the smallest idea that occurred to me.

My family has provided me with moral support and inspiration which have brought me so far in my education. My friends in Pittsburgh (human and feline) have kept me in good spirits. Without them, I would have been unable to stay emotionally well-rounded through my academic endeavors. The Ashars in particular made sure that I remembered to eat, and gave me a power adapter for my personal computer when I accidentally marinated mine in a puddle of yogurt a week before the thesis presentation. Thank you, everyone!

Contents

1 Introduction **1**

2 Background **5**
- 2.1 Substructural Specifications in CLF 5
- 2.2 Semantics-driven Compiler Generation 15

3 S3OS: Separable SSOS **17**
- 3.1 Static vs. Dynamic Content 17
- 3.2 Compilation 24
- 3.3 Virtual Machine 31
- 3.4 Summary 35

4 S4OS: Semicompositional S3OS **39**
- 4.1 Functions 39
- 4.2 Synthesis for S4OS 43
- 4.3 Semicompositionality and Loops 47

5 Conclusion **53**
- 5.1 Contributions 53
- 5.2 S40S for C0 55
- 5.3 Future Work 56
- 5.4 Related Work 57

A Proofs Sketches of Theorems **59**
- A.1 Properties from Chapter 3 59
- A.2 Properties from Chapter 4 66

B $\mathcal{L}$ listings for chapter 3 **69**
- B.1 Support Code 69
- B.2 S3OS-sko 70
- B.3 S3OS-comp 71
- B.4 Simplified S3OS-VM 72

C Higher Order Features **73**
C.1 De Bruijn Indices and Closures 73
C.2 Call by name λ Calculus 74
C.3 Futures 76

Bibliography **79**

Listings

2.1 Result of adding bits . . . 13
2.2 Example of parallel evaluation . . . 13
3.1 SSOS for $\mathcal{L}$ . . . 18
3.2 Sample program fragment . . . 19
3.3 CLF encoding of listing 3.2 . . . 19
3.4 Continuations from listing 3.3 . . . 19
3.5 Assembly for listing 3.4 . . . 19
3.6 `exp/num` without immediates . . . 21
3.7 `exp/num` with immediates . . . 21
3.8 Continuations from listing 3.3 with `f/num` . . . 21
3.9 Assembly for listing 3.8 . . . 21
3.10 `f/op2` before separation . . . 22
3.11 `f/op2` after separation . . . 22
3.12 `tr/let` before separation . . . 23
3.13 `tr/let` after separation . . . 23
3.14 Destinations in expressions . . . 23
3.15 Listing 3.3 with destinations . . . 23
3.16 Separated continuations from listing 3.15 . . . 24
3.17 Assembly for listing 3.16 . . . 24
3.18 Skolemization with Equality . . . 26
3.19 `let` rules with constructed destinations . . . 26
3.20 `ev/let` before freshening . . . 28
3.21 `ev/let` after freshening . . . 28
3.22 `tr/dummy` . . . 28
3.23 Continuations from listing 3.16 after skolemization and freshening . . . 29
3.24 Assembly . . . 29
3.25 After partial erasure . . . 30
3.26 After mode-driven erasure . . . 30
3.27 Persistent continuations . . . 33
3.28 Stop producing continuations . . . 33
3.29 Erase valuables . . . 33
3.30 Disambiguate by matching continuations . . . 33
3.31 Code simplification . . . 34
3.32 `tr/op1` with unused valuables . . . 34

3.33 Simplified `tr/op1` . . . 34
3.34 Simplified continuations . . . 35
3.35 Assembly . . . 35
3.36 S3OS requirement specification . . . 37
4.1 Functions with HOAS . . . 40
4.2 Computing with source entry points . . . 40
4.3 Definition of closures . . . 42
4.4 Functions with closures . . . 42
4.5 `let` statements with closures . . . 43
4.6 Listing 4.2 without HOAS . . . 44
4.7 Arithmetic operations with closures . . . 44
4.8 Skolemized function semantics . . . 45
4.9 Compilation of functions . . . 45
4.10 Virtual instructions for functions . . . 46
4.11 `set` statements . . . 48
4.12 Query that fails due to unconsumed resources . . . 48
4.13 Garbage collected mutable state . . . 48
4.14 Automatic variables on the stack . . . 49
4.15 Functions allocate automatic variables . . . 49
4.16 While loop semantics . . . 50
4.17 Modified statement semantics . . . 51
4.18 Modified function return . . . 51

Chapter 1

Introduction

Compiler generation has been studied for styles of semantic specification such as denotational semantics [SS71], natural semantics [Kah87], Structural Operational Semantics (SOS) [PuDa81], action semantics [Mos96] and modular SOS (MSOS) [Mos99]. The goal of research in this area was to eliminate the performance overhead of interpretation usually associated with interpreters transliterated from semantic specifications. Semantically generated compilers would reduce or eliminate the logistical overhead of developing a separate compiler and proving its correctness. Prior research has been most successful at generating compilers from denotational semantics and action semantics [Lee87] [Ørb94] [DV96].

This dissertation explores semantics-driven compiler generation for Substructural Operational Semantics (SSOS). SSOS, developed by Pfenning et al [Pfe04, Pfe06, PS09] combines ideas from SOS and linear logic programming to express semantic specifications of programming languages as state transitions in abstract machines. Prior work has demonstrated that SSOS is suitable for modular specification of concurrent, non-deterministic and imperative features. We aim to bring the benefits of compiler generation to SSOS also.

At a high level, the state of the abstract machine modeled by SSOS is a multiset of resources corresponding to the context in intuitionistic linear logic, and transitions in SSOS are based on inference rules in linear logic. An inference rule can be applied if the resources matched by its antecedent are present in the multiset. Once applied, the rule rewrites the multiset to replace the resources matched in the antecedent with those produced in the consequent of the rule. Thus, the state of the machine evolves through the repeated application of rewrite rules until some desired final state is reached, or no rule in the specification matches the state. The correspondence between SSOS specifications and multiset rewrite rules is due to Cervesato et al in [CS09].

SSOS specifications typically contain two types of rules – *active* and *latent*. An active rule matches exactly one active resource in the multiset. The action of such a rule can be visualized as follows:

$$\Delta, \mathsf{active} \rightsquigarrow \Delta, r_1, r_2, ..., r_n$$

The abstract machine transitions from the state on the left to the state to the right, and Δ encodes the parts of the multiset that are not matched by the transition rule. For brevity, we can elide Δ and write this same transition as:

$$\mathsf{active} \rightsquigarrow r_1, r_2, ..., r_n$$

Latent rules match on multiple resources, one of which is latent and the rest of which are *passive*:

$$\mathsf{passive}_1, \mathsf{passive}_2, ..., \mathsf{passive}_m, \mathsf{latent} \rightsquigarrow r_1, r_2, ..., r_n$$

We say that the latent resource is a suspended computation which can be triggered by the availability of suitable passive resource.

Our thesis statement is given in three parts, of which the first follows:

Thesis Statement 1. *The substructural operational semantics of a programming language can be stated in such a way that the latent resources are of static provenance, and passive resources are dynamic values. In such a form, the latent resources correspond to instructions synthesized from the program, and passive resources correspond to operands to these instructions.*

By static provenance, we mean that latent resources are computed from the program text only and do not contain any values computed while running the program. Dynamic values are those which are computed when running the program on its arguments. We contribute a variant of SSOS called S4OS which separates static and dynamic content into latent and passive resources respectively, allowing us to treat passive resources as instructions (in the sense of instruction set architectures for machines). The instructions do not belong to any particular instruction set. They form a virtual instruction set that is characteristic of the language whose semantics we are working with.

Given the S4OS for a programming language which clearly identifies static and dynamic content, we wish to synthesize a compiler for the language. This brings us to the second part of the thesis statement:

Thesis Statement 2. *Given a separable SSOS specification for a programming language, the latent resources/instructions for a program can be computed without referring to the program's inputs or dynamic values. We can do so by logically approximating the behavior of the specification.*

Towards this statement, we contribute specific ways in which the persistence modality and Skolemization can be applied to the S4OS to approximate its behavior. Our techniques are similar to those described by Simmons and Pfenning in [SP09]. We also contribute *mode-driven erasure* which is essential to remove references to dynamic content during compilation. We collectively refer to these transformations as linear logical approximations (term borrowed from [SP09]).

Once we have collected these instructions for a program, we need a way to execute the instruction set. We also need a concise specification for the virtual instruction set that guides the process of targeting a machine implemented in hardware. This brings us to the third and final part of the thesis statement:

Thesis Statement 3. *A separable SSOS specification can be used to synthesize a virtual machine that executes the virtual instruction set characteristic of the object language. The virtual machine serves as a specification of the instruction set, and it does not refer to the original program text.*

In support of the above statement, we demonstrate how to use persistence, skolemization and erasure of static content to synthesize a virtual machine. We do not address the final step of converting the virtual instruction set to other instruction sets in this paper, but we discuss possibilities in the conclusion.

We tested our work by specifying the SSOS for C0, a safe subset of C used to teach imperative programming at Carnegie Mellon University [Pfe11]. The design space of C0 is too large to concisely illustrate our techniques, so we illustrate carefully selected features from C0 in the context of a smaller language.

We demonstrate our work with SSOS encodings in CLF (Concurrent Logical Framework), a logical framework based on lax linear logic [WCPW03] [CPWW03]. We chose CLF for this dissertation because it has an implementation called Celf that is in active development. However, our contributions are not specific to CLF encodings and can likely be ported to encodings in other logical frameworks such as the recently developed SLS [Sim12].

The remainder of this dissertation is organized as follows: Chapter 2 provides essential background information on SSOS encodings in CLF, and prior work in compiler generation. Chapter 3 walks through a specification style called S3OS, a style of specification simpler than S4OS but still capable of demonstrating compiler and VM generation for simple language features. It is used to synthesize a compiler and VM for a very small imperative language called $\mathcal{L}$. Chapter 4 further develops S3OS into S4OS, which can be used to extend $\mathcal{L}$ with more complex features inspired by C0. Chapter 5 concludes the dissertation with a summary of contributions, related work and future work.

Chapter 2

Background

This chapter provides required background on SSOS and semantics-driven compiler generation. It focuses on defining recurring examples and terminology. Section 2.1 introduces substructural specifications in CLF. All the examples are tested using Celf, the reference implementation of CLF. Section 2.2 surveys existing semantics-driven compiler generation techniques for useful terminology and concepts.

2.1 Substructural Specifications in CLF

We implement compiler generation for language encodings and SSOS in CLF, a logical framework based on intuitionistic lax linear logic. A comprehensive reference to CLF and Celf can be found in [WCPW03, CPWW03]. SSOS was explored and developed by Pfenning et al. in [CPWW03, Pfe04, PS09, Sim12].

We avoid presenting the details of the underlying logic and proof theory in favor of a high level, example driven perspective. Readers may find it helpful to have some prior experience with intuitionistic linear logic and logic programming. A detailed tutorial of intuitionistic linear logic and logic programming may be found in [Pfe12b]. Readers who are already familiar with CLF and SSOS may find it useful to skim through the section and read the sample CLF signatures, which will be reused in chapter 3 to present original work.

2.1.1 Encoding Abstract Syntax

In this section we introduce a simple imperative language called $\mathcal{L}$ and encode it in CLF. The following is the grammar of $\mathcal{L}$.

$\langle exp \rangle$ ::= $\langle numeric\text{-}constant \rangle$ | $\langle exp \rangle$ $\langle op \rangle$ $\langle exp \rangle$

$\langle op \rangle$ ::= `+` | `x` | ...

$\langle stm \rangle$ ::= `return` $\langle exp \rangle$ `;` | `let` $\langle var \rangle$ `=` $\langle exp \rangle$ `;` $\langle stm \rangle$

$\langle prog \rangle$ ::= $\langle stm \rangle$

The following is an example of a $\mathcal{L}$ program:

```
let a = 6;
let b = 7;
return (a + b) * (a + b);
```

This program computes the square of the sum of two numbers. $\mathcal{L}$ supports the manipulation of numeric values using binary infix arithmetic operations. These arithmetic computations may be nested. Additionally, expressions may be sequenced in statements. There are two types of statements: `let` statements which bind expressions to variables, and `return` which is the last statement in the program and determines the result of the program.

We need to define an abstract syntax for $\mathcal{L}$ so that we may encode programs such as the given example in CLF. The classes of source terms in the abstract syntax are expressions, operators, statements, constants and variables. We define expressions, operators and statements as CLF types below:

```
op  : type. % operators
exp : type. % expressions
stm : type. % statements
```

The listing above is called a *signature*. A signature is a list of CLF type and term declarations. Every type or term declaration is delimited at the end by a period. Any text following a `%` is a comment. Let us define numeric constants to be binary words. For this, we need two more syntactic classes: bits and binary words (lists of bits).

```
bit : type.
bin : type.
```

Next, we need to define how to construct valid numeric constants and expressions. Let us begin with bits and binary words:

```
b0 : bit.
b1 : bit.
```

These are term definitions (unlike the type definitions stated previously). Their types indicate how they can be used to construct other terms. There are two nullary constructors for bits corresponding to 0 and 1.

```
bin/e : bin.
bin/b : bin -> bit -> bin.
```

Binary words are defined inductively. There is a nullary constructor which can be used to construct an empty word. `bin/b` can be applied to an existing `bin` and `bit` to produce another term. `->` is persistent implication. Note that the type `bin -> bit -> bit` is parenthesized as `bin -> (bit -> bit)`. We can interpret the use of this constructor as the act of appending a new least significant bit to an existing binary word. Let us see how these constructors are used to create terms:

```
w6 : bin = (bin/b (bin/b (bin/b (bin/b bin/e b0) b1) b1) b0).
w7 : bin = (bin/b (bin/b (bin/b (bin/b bin/e b0) b1) b1) b1).
```

`b6` and `b7` are definitions of the binary numeric constants 6 and 7 as 4-bit words. Definitions in CLF are given using `=`. They are defined using existing constructors.

There are other plausible inductive definitions of binary words. The base case could be one bit instead of an empty word, in which case the constructor would be `bin/e : bit -> bin`.

Also, we could have inductively constructed binary words by adding a new most significant bit, in which case the constructor would be `bin/b : bit -> bin -> bin`. We will not use any of these other formulations.

Next, we define constructors for operators, expressions and statements.

```
op/+ : op.
op/x : op.    % can't say op/* because * is reserved in Celf

exp/num : bin -> exp.
exp/op  : exp -> op -> exp -> exp.
```

Most of these constructors are straightforward. We define two nullary constructors for operators corresponding to addition and multiplication. More can easily be added. There are two constructors inductively defining expressions – the first constructs expressions from numeric constants, and the second composes two subexpressions with an operator. For example, the following are encodings of $\mathcal{L}$ expressions in CLF:

```
e6+7   : exp = (exp/op (exp/num w6) op/+ (exp/num w7)). % (6 + 7)
e6+7sq : exp = (exp/op e6+7 op/x e6+7).                 % (6 + 7) * (6 + 7)
```

Statements are also inductively defined.

```
stm/return : exp -> stm.
stm/let    : exp -> (exp -> stm) -> stm.
```

The base case is the return statement which is constructed from an expression to evaluate and return as a result. The `stm/let` is perhaps the most interesting constructor defined so far. A let statement must bind an expression to a variable that is within scope in the next statement. However, the constructor does not construct the statement from an expression, a variable and the next statement. We have not even defined a separate CLF type for variables. Let us encode a few statements to see how we can get by without separately defined variables:

```
a+b : stm = (stm/let (exp/num w6) (\!a.                 % let a = 6;
              (stm/let (exp/num w7) (\!b.               % let b = 7;
                (stm/return (exp/op !a op/+ !b)))))).   % return a + b;
```

This example introduces more CLF syntax. `(\!a. <CLF term>)` is the syntax for λ-abstraction which in this case binds the variable `a`. The `!` is the persistence modality from linear logic. The use of the `!` modality indicates that the body may refer to the variable `a` an arbitrary number of times, without the constraints of linearity. In the example, we encode a statement that requires an expression to be bound within its scope using a λ-term that binds a CLF variable with type `exp`. The example demonstrates that CLF variables can be used to encode object language variables that have lexical scope. The technique of using logical framework variables to encode object language variables is called Higher Order Abstract Syntax (HOAS) [HL78, HHP93, PE88].

The example should elucidate the purpose of the type of `stm/let`. The first `exp` refers to the expression that must be evaluated and bound. The `(exp -> stm)` is the type of a statement that requires an `exp` to be lexically bound within its scope. We have now provided all the CLF definitions necessary to encode any $\mathcal{L}$ program.

2.1.2 Substructural Operational Semantics (SSOS)

This section describes how to compute with $\mathcal{L}$ programs encoded in CLF by defining the SSOS for the language. Computation in CLF is primarily expressed as logical inference and proof search. Proof search in logic programming environments like Prolog involves the use of deductive reasoning to determine the truth or falsehood of propositions. Proof search in CLF additionally constructs a proof term for a given type using valid inferences. Inferences can either be forward or backward. For example, let us implement a ripple adder using proof search.

```
bin/add : bin -> bin -> bit -> bin -> bit -> type.
#mode bin/add + + + - -.
```

The first line declares `bin/add` as a *type family* which relates two input binary words, a carry-in bit, an output binary word, and a carry-out bit. Since we mean to establish an input-output relationship between the terms in the type family, we also supply a *mode declaration*[1]. The first three + signs declare that the first three terms, i.e. the two binary words and the carry-in bit are inputs. The next two - signs declare that the remaining binary word and carry-out bit are outputs. The input-output relationship is not a part of the type system proper, but is designed to provide safety constraints. When a type family is given a mode declaration, Celf performs *mode checking* on all occurrences of a type of that family in the signature. We will give an example of this shortly. In order to implement a ripple adder, we need a one bit full adder:

```
bit/add : bit -> bit -> bit -> bit -> bit -> type.
#mode bit/add + + + - -.
```

This definition should be similar to that of the ripple adder. `bit/add` relates two input bits, a carry-in bit, one output bit and a carry-out bit. In order to compute using proof search, we need to provide a way to find proof terms that satisfy the type. For this purpose, we define terms in the signature that describe how a full adder relates its inputs and outputs.

```
bit/add/000 : bit/add b0 b0 b0 b0 b0.
bit/add/001 : bit/add b0 b0 b1 b0 b1.
bit/add/010 : bit/add b0 b1 b0 b0 b1.
bit/add/100 : bit/add b1 b0 b0 b0 b1.
bit/add/011 : bit/add b0 b1 b1 b1 b0.
bit/add/101 : bit/add b1 b0 b1 b1 b0.
bit/add/110 : bit/add b1 b1 b0 b1 b0.
bit/add/111 : bit/add b1 b1 b1 b1 b1.
```

The terms are very similar to the truth table for full adders. All terms declared in a signature are part of the *context* of persistent assumptions, so that proof search can find these terms and use them to construct a proof[2]. In Celf, we can initiate proof search using a `query`. For example:

```
#query * 1 * 1
bit/add b1 b0 b0 S C.
```

[1]The concept of mode and its significance for termination of prolog programs were developed in [War77] and [AP93], amongst other works. Mode checking was first implemented in a higher order setting in Elf [RP96] and later in Celf.

[2]We refer to the set of assumptions that can be used to construct a proof term for a type as *the context*. A complete discussion of the context is beyond the scope of this dissertation.

The first line directs Celf to make one attempt at solving the query and print out one solution. The second line is the type for which the query must construct a proof term. The capitalized variables S and C are to be instantiated by Celf during proof search. We follow the convention from Prolog of capitalizing variables instantiated during proof search by Celf. Constants and bound variables use lowercase only. The query produces the following output:

```
Solution: bit/add/100
 #C = b1
 #S = b0
```

Proof search found one of the terms we defined in the context that satisfied all the constraints present in the input positions. Now let us step through the implement of a ripple adder for words of equal length using proof search:

```
bin/add/nil :
  bin/add bin/e bin/e C bin/e C.

bin/add/bit :
  bin/add (bin/b N1 B1) (bin/b N2 B2) C (bin/b N3 B3) C''
  o- bit/add C B1 B2 C' B3
  o- bin/add N1 N2 C' N3 C''.
```

The first clause states the base case, i.e. the sum of two empty words is an empty word. The carry-out bit is the same as the carry-in bit. `bin/add/nil` provides a useful illustration of mode checking. Mode checking ensures that the terms in the output positions can all be determined from the terms in the input positions during proof search. The first output is constructor from the signature, and the second output position is a variable that is also present in an input position. Therefore `bin/add/nil` passes mode checking and is said to be well-moded. Mode checking ensures that the type of `bin/add/nil` is always completely known for any instantiation of variables in input positions.

The second clause `bin/add/bit` is more complex and introduces new CLF syntax. The `o-` connective is the linear implication connective `-o` from linear logic written backward, i.e. `(B o- A)` must be read as "`A` implies `B`". The implication is written backward to continue the tradition of Prolog. For the purpose of proof search, the consequent of the implication is considered the *goal* that we may wish to prove, and the antecedent is considered the *subgoal* to which the goal is reduced. In the case of `bin/add/bit`, the goal is to add two nonempty binary words using the ripple carry method. This goal is reduced to the subgoals of adding the least significant bits using the one bit full adder and recursively adding the remaining bits of the input words using ripple carry.

Let us informally reason about the well-modedness of `bin/add/bit`. The variables occurring in the input positions of a subgoal are required to be variables that occur in the input position of the goal (e.g. `B1` and `B2`), or in the output position of a previous subgoal (e.g. `C'` and `B3`). Variables that occur in the output position of a goal must either occur in an input position of the same goal, or the output position of a subgoal. Therefore, the terms in the output positions (and consequently the full type) is completely known for any instantiation of `bin/add/bit`.

Clauses such as `bin/add/nil` and `bin/add/bit` are called *rules* because of the role they play in directing proof search. These rules are used in *backward chaining* because of the way in which they reduce goals to subgoals which may ultimately be proven in terms of atoms that

exist in the signature. Since atoms in the signature are persistent, the use of linear implication in the given rules is not significant. The following is another example of a Celf query and a proof constructed by proof search.

```
#query * 1 * 1
bin/add w6 w7 b0 S C.


Solution:
  bin/add/bit (bin/add/bit (bin/add/bit
    (bin/add/bit bin/add/nil bit/add/100) bit/add/111) bit/add/011) bit/add/001
 #C = b0
 #S = bin/b !(bin/b !(bin/b !(bin/b !bin/e !b1) !b1) !b0) !b1
```

Forward Chaining

Next, we wish to illustrate *forward chaining* by implementing SSOS transition rules for $\mathcal{L}$ expressions and statements. In forward chaining, we begin with known facts and follow implications in the forward direction until we either reach a desired set of facts or no further inferences can be made. Forward chaining is ideal for implementing abstract machines. States can be encoded as multisets of *linear types*. Linear implication can be used to encode transition rules. The following examples illustrate forward chaining rules:

```
valuable : type.    vbl/e : exp -> valuable.    vbl/s : stm -> valuable.
value    : type.    val/e : exp -> value.

dest : type.
eval : valuable -> dest -> type.       #mode eval - -.
ret  : value -> dest -> type.          #mode ret  - -.
```

$\mathcal{L}$ has expressions and statements, but we wish to reason about them both as valuable terms, i.e. terms that can be evaluated by the abstract machine. So, we declare a type called `valuable` and provide constructors that accept expressions and statements. Upon evaluation, the machine may produce values, so we declare another type called `value`. All $\mathcal{L}$ values are expressions, so we have only one constructor for values at present. The meaning of `dest` will become clear shortly.

`eval` and `ret` types each encode some part of the state of the abstract machine. `eval` stands for "evaluation". `eval V W` states that the machine is evaluating the valuable term `V` with the goal of returning the computed result at a destination `W`. `ret` stands for "return". `ret V W` states that the machine has returned a value `V` at a destination `W`. Destination are basically identifiers for assumptions in the context, and their uses are elucidated in the following examples. The following is an example of a transition rule encoded as implication using evaluations and returns:

```
ev/num : eval (vbl/e (exp/num N)) W
         -o { ret (val/e (exp/num N)) W }.
```

The rule states that evaluation of a numeric expression immediately returns the same number as a value. This rule introduces more CLF syntax – the { braces } that are placed around the consequent of the implication. These braces correspond to the lax modality from lax logic. A forward chaining rule is logically distinguished from a backward chaining rule by the presence

of the lax modality in the consequent[3]. To further stylistically highlight that the rule uses forward inference, we write the linear implication in the forward direction (`-o` instead of `o-`).

We have previously mentioned the use of persistent assumptions in the context during backward proof search. `ev/num` is an example of a rule that operates on linear assumptions in the context. A linear occurrence of `eval` in the context can be consumed by this rule to produce a linear `ret` in its place. The context of linear assumptions is analogous to a multiset and the rule is analogous to a multiset rewrite rule, an observation due to Cervesato et al [CS09]. Since linearity can be used to reason about consumption and production, linear assumptions in the context are also called *ephemeral resources*.

Next we need to evaluate arithmetic operations. Suppose we wish to evaluate the two operands sequentially from left to right, and then operate on the results.

```
ev/op : eval (vbl/e (exp/op E1 B E2)) W
        -o { eval (vbl/e E1) ... ? }.
```

This rule is not complete. We know that we need to start evaluation of the first subexpression, which is what the rule does so far. However, we need to evaluate the subexpression in such a way that its result is not confused with the result of the top level expression or any other subexpression. We also need to save the operator and the second subexpression so that we can resume computing once we get a result for the first subexpression. We solve the first problem as follows:

```
ev/op : eval (vbl/e (exp/op E1 B E2)) W
        -o { Exists x : dest. eval (vbl/e E1) x * ... ? }.
```

The rule introduces the use of another CLF feature – the existential quantifier. We use the existential quantifier to create a new destination, and evaluate the subexpression at this new destination. The logic ensures that two existentially quantified variables generated at different sites are not equal, so we exploit the quantifier as a symbol generator. Now we can distinguish between the results of different evaluations because we identify them by fresh destinations.

To solve the problem of saving computations to resume them, we define a few more types:

```
frame : type.
cont  : frame -> dest -> dest -> type. #mode cont - - -.
```

`cont` stands for "continuation". `cont F X W` can be used to encode a suspended computation that waits for a result to become available at the destination `X` and continues evaluation to produce a result at `W`. `F` is the frame, which actually contains saved values or valuable terms. Naturally, we need to define frame constructors which can construct frames from the terms that have to be saved.

```
f/op1 : op -> valuable -> frame.
ev/op : eval (vbl/e (exp/op E1 B E2)) W
        -o { Exists x. eval (vbl/e E1) x *
             cont (f/op1 B (vbl/e E2)) x W }.
```

`f/op1` allows the rule to store the second subexpression and operator into a continuation and proceed to evaluate the first subexpression. The continuation is said to be *waiting* on the destination `x`. This rule introduce another ubiquitous piece of CLF syntax – the `*` connective. `*` is the

[3]A complete discussion of the lax modality is beyond the scope of this dissertation. Please consult [WCPW03, CPWW03], or [Pfe12b] for details.

multiplicative conjunction from linear logic. Now we need to specifying how computation can continue once this subexpression finishes evaluating.

```
f/op2 : value -> op -> frame.
tr/op1 : ret V1 X * cont (f/op1 B E2) X W
         -o { Exists x. eval E2 x * cont (f/op2 V1 B) x W }.
```

Unlike the previously given rules (evaluation rules), this rule is a trigger rule because it describes how to trigger a suspended computation. The result in the antecedent of the rule is matched to the waiting continuation using its destination. This style of relating computations using destinations is called linear destination passing style [Pfe04]. The newly produced continuation saves the result of the first subexpression while waiting for the result of the second subexpression. The following rule describes what happens when both results are available and we wish to add them:

```
tr/op2/add : ret (val/e (exp/num N2)) X *
             cont (f/op2 (val/e (exp/num N1)) op/+) X W *
             !bin/add N1 N2 b0 N3 _
             -o { ret (val/e (exp/num N3)) W }.
```

When Celf attempts to match this rule to the context, it has to switch from forward chaining to backward proof search to run the ripple adder that we previously defined. Once the addition is completed, the rule returns the result and Celf continues with forward chaining. It is not strictly necessary to have used the `!` modality with `bin/add`. Nevertheless we used the modality as a matter of style because it forces the proof term to not use any resources in the linear context. This is analogous to a guarantee that addition does not mutate any of the state of the machine.

Celf also needs to mode check the use of `bin/add`. This requires a mode declaration for `ret`, `cont` and `eval`. Each of these resources is only used in a forward chaining setting. Any rule that matches them in the context needs to know all of the terms in each resource. Therefore, we have provided mode declarations that classify all the terms in returns, continuations and evaluations as outputs. Now we know that the use of `bin/add` in `tr/op2/add` is well-moded because outputs that are first matched in a return and a continuation are fed as inputs to the ripple adder.

How to start evaluating an expression in Celf? The query feature initiates backward chaining proof search. We need to design a query that will load an evaluation into the context, and switch to forward chaining. The following query meets these requirements:

```
w : dest.
#query * 1 * 1
eval (vbl/e e6+7) w -o { ret V w }.
```

We direct Celf to prove an implication. In order to prove an implication, we need to assume the antecedent. Therefore, Celf loads the evaluation in the antecedent into the context. The goal is a lax type, and Celf switches to forward chaining to prove the goal. Listing 2.1 contains the results of the query. Every forward chaining step is recorded by let binding the result of applying one of the forward inference rules.

Concurrency

An advantage of SSOS is the ease with which one can express concurrency and parallelism. For instance, suppose we wished to evaluate the two operands in arithmetic operations in parallel.

```
Solution: \X3. {
    let {[!x, [X4, X5]]} = ev/op X3 in
    let {X6} = ev/num X4 in
    let {[!x_1, [X7, X8]]} = tr/op1 [X6, X5] in
    let {X9} = ev/num X7 in
    let {X10} = tr/op2/add [X9, [X8,
      !(bin/add/bit (bin/add/bit (bin/add/bit
         (bin/add/bit bin/add/nil bit/add/100) bit/add/111) bit/add/011)
         bit/add/001)]] in X10}
 #V = val/e !(exp/num !(bin/b !(bin/b
        !(bin/b !(bin/b !bin/e !b1) !b1) !b0) !b1))
```

Listing 2.1: Result of adding bits

```
cont2     : frame -> dest -> dest -> dest -> type. #mode cont2 - - - -.
f/op-par : op -> frame.

ev/op : eval (vbl/e (exp/op E1 B E2)) W
        -o { Exists x1. eval (vbl/e E1) x1 *
             Exists x2. eval (vbl/e E2) x2 *
             cont2 (f/op-par B) x1 x2 W }.

tr/op-par/add : ret (val/e (exp/num N1)) X1 *
                ret (val/e (exp/num N2)) X2 *
                cont2 (f/op-par op/+) X1 X2 W *
                !bin/add N1 N2 b0 N3 _
                -o { ret (val/e (exp/num N3)) W }.
```

Listing 2.2: Example of parallel evaluation

The rules in listing 2.2 should achieve the desired effect. We simply need to extend SSOS with continuations that can wait for multiple results.

Forward chaining in Celf is implemented as *nondeterministic committed choice*. If multiple rules are applicable to a given context, as may be the case when evaluating concurrent computations, Celf will nondeterministically choose one rule and commit to it. If forward chaining fails, Celf does not backtrack to consider another applicable rule. Due to this behavior, forward chaining is not exactly proof search, though it uses only inferences that are valid in the logic. The backward chaining operations (such as the ripple adder) continue to function as atomic operations and have backtracking available to them.

The lack of backtracking in forward chaining allows us to improve the efficiency of computations that have deterministic parallelism. It also allows us to run the SSOS specification as a genuinely nondeterministic interpreter when encoding nondeterministic languages. We do not illustrate compilation for any nondeterministic language features in this dissertation. However, nondeterminism must be kept in mind because it affects the way in which we synthesize compilers and virtual machines from SSOS. It also affects the quality of compiled code.

Substitution

Next, consider the semantics for $\mathcal{L}$ statements. This gives us an opportunity to work with substitution in CLF. In order to evaluate a let statement, we first need to evaluate the expression that needs to be bound:

```
f/let : (exp -> stm) -> frame.
ev/let : eval (vbl/s (stm/let E S)) W
         -o { Exists x. eval (vbl/e E) x * cont (f/let S) x W }.
```

This rule should be straightforward. We start evaluating the expression, and suspend the following statements in a continuation. Once the value becomes available, we need to bind it in a way accessible to the remaining computation. Since we used CLF variables to encode $\mathcal{L}$ variables, the returned value can simply be substituted for the variable.

```
tr/let : ret (val/e !V) X * cont (f/let (\!e. S !e)) X W
         -o { eval (vbl/s (S !V)) W }.
```

The last remaining $\mathcal{L}$ feature to be specified is the return statement. The return statement is syntactic sugar that allows us to place an expression at the end of a list of statements. We can simply peel away this syntactic sugar and evaluate the expression:

```
ev/return : eval (vbl/s (stm/return E)) W -o { eval (vbl/e E) W }.
```

We didn't need to save any terms, so we did not produce any continuations.

Summary of SSOS requirements

The style of specification illustrated in this section, in which an abstract machine is encoded using rewrite rules in substructural logic is called Substructural Operational Semantics. In this dissertation, we use the term SSOS to more specifically refer to a *defunctionalized* specification that uses evaluations returns and continuations. By defunctionalized, we mean that transition rules are not used as first class resources – they must not be consumed or produced by other transition rules. Our contributions are not directly applicable to a functional style of specification such as the one presented by Cervesato et al. in [CPWW03], unless it is first defunctionalized by a pass such as the one presented by Simmons in [Sim12].

We also require an SSOS specification to observe a separation between *active*, *latent* and *passive* computations [PS09]. Evaluations are called active computations because they initiate evaluation without depending on any other resource in the context. As a result evaluation rules can be applied asynchronously. Returned values are called passive because they do not inherently contain any remaining computation to be performed. Continuations are latent because they are triggered or activated by passive resources in trigger rules. We do not permit rules to match evaluations and returns in the antecedent of the same rule. Similarly, evaluations and continuations cannot be present in the antecedent of the same rule. This restriction allows us to clearly classify all rewrite rules as evaluation rules or trigger rules.

Finally, we relate the classification of rewrite rules back to the thesis statement in chapter 1. Evaluation rules are examples of transitions that were called *active transitions*, while trigger rules are examples of *latent transitions*. Chapter 3 explains how to express the computational content of a specification using only latent transitions, and compile with these latent transitions. This summary concludes the introduction to CLF and SSOS.

2.2 Semantics-driven Compiler Generation

Prior work has explored semantics-driven compiler generation for styles of semantic specifications including denotational semantics, natural semantics, structural operation semantics and action semantics. The techniques used for each of these styles inherits from a common ancestor – generating compilers from interpreters. It is possible to transliterate each style of semantic specification into an interpreter in a straightforward manner. In fact, section 2.1 explains how to encode the SSOS of a programming language in CLF and run it as an interpreter.

The next step is to synthesize a compiler from the interpreter. A variety of approaches have been taken over the years, but we particularly survey the literature on partial evaluation for terminology and concepts that are reused in S3OS and S4OS.

In the 1970s, Futamura explained the correspondence between compilers and partially evaluated interpreters with identities that later came to be known as the Futamura projections [Fut99]. The identities are state below.

Given a program p with *static* inputs s and *dynamic* inputs d, we can specialize p to the static inputs s by applying a partial evaluator α. The following identity relates the specialized program to the original program.

$$p(s, d) = \alpha(p, s)(d)$$

The result of specialization $\alpha(p, s)$ is called the *residual* program. Suppose p is actually an interpreter int for a programming language $\mathcal{L}$, s a program prog to be interpreted, and d the arguments args to the program. The following are the Futamura projections:

$$\mathsf{int}(\mathsf{prog}, \mathsf{args}) = \alpha(\mathsf{int}, \mathsf{prog})(\mathsf{args})$$

$$\alpha(\mathsf{int}, \mathsf{prog})(\mathsf{args}) = \alpha(\alpha, \mathsf{int})(\mathsf{prog})(\mathsf{args})$$

$\alpha(\alpha, \mathsf{int})$ has the signature of a compiler. In other words $\alpha(\alpha, \mathsf{int})$ accepts text in $\mathcal{L}$ and outputs another residual program P. P is the compiled version of prog. Note the underlying assumption that the interpreter specification is written in some meta language M that can be partially evaluated, and the partial evaluator must also be in M and designed to specifically allow efficient specialization. Also note that we follow the tradition of partial evaluation and refer to terms derived from the program text as *static* parts of the interpreter. The runtime arguments to the program and values obtained from the execution environment are called *dynamic*.

Synthesizing a good compiler is not as simple as running a partial evaluator on an interpreter. Poorly written interpreters can produce bloated residual programs or entirely fail to specialize to an input program. Some styles of semantic specification are also more susceptible to poor specialization that others. Neil Jones identifies some qualities that make an interpreter suitable for specialization in [Jon96]. We present a few that we reuse in S3OS and S4OS:

Bounded Static Variation

Any term in a program can be said to satisfy *Bounded Static Variation (BSV)* if the term can be proven to evaluate to a finite number of possible values, and all the possible values can be computed using only the static inputs of the program. A partial evaluator can evaluate all terms

satisfying BSV during specialization, and leave the remaining terms to be evaluated later in the residual program.

Determining when a term satisfies BSV can be quite complicated, and is the subject of an analysis called Binding-time Analysis (BTA). BTAs usually require some participation from the programmer, such as annotations indicating which terms satisfy BSV. More recent work has developed type systems that allow us to express these annotations in a machine checkable form [Dav96][PD01].

Good separation of static and dynamic terms in the interpreter is essential for a BTA to identify all source terms. An interpreter may mingle static and dynamic terms by storing them in shared data structures or by substituting terms of dynamic origin into the program text. If the BTA does not successfully determine whether syntactic terms in the interpreter satisfy BSV, then the synthesized compiler would have to defer the translation of some of these terms to runtime to ensure termination of compilation. As a result, we would be unable to completely eliminate interpretation overhead.

If we design interpreters from the ground up for easy specialization, we should strive to make them *compositional* or *semicompositional*. These are two important classes of interpreters identified in [Jon96] that ensure separability and simplify the BTA.

Compositional Interpreters

Suppose we have an interpreter which interprets a program by recursively invoking the interpreter on terms derived from the program text. Most of the semantic specification styles discussed so far can be transliterated into such an interpreter. The given interpreter is said to be compositional if the syntactic (static) argument passed to each recursive call is a proper syntactic substructure of the static argument passed to the enclosing call. For instance, the $\mathcal{L}$ implementation in 2.1 would be compositional if we excluded `let` statements.

By induction, we can easily establish that the syntactic terms must satisfy BSV. Any synthesized compiler that preserves the same recursive structure, either through semantic correspondence or by unfolding recursive calls during specialization is guaranteed to terminate. Since every source term on which we would invoke the interpreter is passed to the compiler ahead of time, the overhead of interpretation can effectively be reduced or eliminated.

Semicompositional Interpreters

Semi-compositional interpreters are more flexible than compositional interpreters. They permit recursive calls with syntactic arguments that only have to be syntactic substructures of the entire program to be interpreter. This allows the size of the term to not shrink, or even grown in a recursive call with respect to the enclosing call.

Relaxing the size metric allows us to specify language features such as loops and functions which involve non-local control flow or repetition. We can still establish that the syntactic terms satisfy BSV because there can only be a finite number of subterms given program text of finite size.

Chapter 3

S3OS: Separable SSOS

In this chapter, we aim to develop some intuition about compiler and VM generation for SSOS using the specification of $\mathcal{L}$, a simple language that combines the features presented in section 2.1. We explain with the aid of well illustrated examples of programs in $\mathcal{L}$ why SSOS as demonstrate in section 2.1.2 doesn't capture all of the requirements of semantics-driven compiler generation. The examples are accompanied by revisions to the SSOS methodology and revisions to the $\mathcal{L}$ specification to conform to the new methodology. The new style of specification is called Separable SSOS, or S3OS. S3OS, though not expressive enough to encode many programming language features, builds an important cognitive base for understanding S4OS and significantly helps substantiate the thesis statement.

The remainder of this chapter is roughly organized according to the three parts of the thesis statement. Section 3.1 explains how S3OS moves most of the computational content of the specification into latent transitions and separates static from dynamic content. Section 3.2 explains how to use approximations such as persistence, Skolemization and mode-driven erasure on S3OS to synthesize a compiler for $\mathcal{L}$. The specifications synthesized in this section are S3OS-sko and S3OS-comp. Section 3.3 explains how program text can be removed from compiled continuations, allowing us to execute them in a compact virtual machine. S3OS-JIT, S3OS-VM and the simplifier are synthesized in this section. We close the chapter with a high level summary of all the techniques.

For reference throughout this chapter, Listing 3.1 provides all the relevant parts of the $\mathcal{L}$ specification.[1].

3.1 Static vs. Dynamic Content

Consider the program fragment in listing 3.2 whose CLF encoding is given in listing 3.3.[2] It computes a result by operating on a number `x` that has not been defined. Suppose we wish to

[1]The backward chaining rules for addition have been elided because they are irrelevant to the compilation methodology which is exclusively concerned with the forward inference rules.

[2]Here, boldface is used to identify the CLF variables that are used to encode object language variables. In general, we use boldface in figures to draw attention to subtle or important points of comparison, be they differences or similarities.

```
bit : type.    b0 : bit.       b1 : bit.
bin : type.    bin/e : bin.    bin/b : bin -> bit -> bin.

bin/add : bin -> bin -> bit -> bin -> bit -> type. #mode bin/add + + + - -.
% bin/add rules omitted.

op : type.    op/+ : op.    % ... more operators if desired

exp     : type.
exp/num : bin -> exp.
exp/op  : exp -> op -> exp -> exp.    % exp1 op exp2

stm        : type.
stm/return : exp -> stm.
stm/let    : exp -> (exp -> stm) -> stm.    % let x = exp1 in stm2

valuable : type.    vbl/e : exp -> valuable.    vbl/s : stm -> valuable.
value    : type.    val/e : exp -> value.

frame : type.
f/op1 : op -> valuable -> frame.
f/op2 : value -> op -> frame.
f/let : (exp -> stm) -> frame.

dest : type.
eval : valuable -> dest -> type.       #mode eval - -.
cont : frame -> dest -> dest -> type. #mode cont - - -.
ret  : value -> dest -> type.          #mode ret  - -.

ev/num : eval (vbl/e (exp/num N)) W
         -o { ret (val/e (exp/num N)) W }.

ev/op : eval (vbl/e (exp/op E1 B E2)) W
        -o { Exists x. eval (vbl/e E1) x *
             cont (f/op1 B (vbl/e E2)) x W }.

tr/op1 : ret V1 X * cont (f/op1 B E2) X W
         -o { Exists x. eval E2 x * cont (f/op2 V1 B) x W }.

tr/op2/add : ret (val/e (exp/num N2)) X *
             cont (f/op2 (val/e (exp/num N1)) op/+) X W *
             !bin/add N1 N2 b0 N3 _
             -o { ret (val/e (exp/num N3)) W }.

ev/return : eval (vbl/s (stm/return E)) W -o { eval (vbl/e E) W }.

ev/let : eval (vbl/s (stm/let E S)) W
         -o { Exists x. eval (vbl/e E) x * cont (f/let S) x W }.

tr/let : ret (val/e !V) X * cont (f/let (\!e. S !e)) X W
         -o { eval (vbl/s (S !V)) W }.
```

Listing 3.1: SSOS for $\mathcal{L}$

```
% x undefined
let y = x + 5;
let z = y + y;
return z;
```

Listing 3.2: Sample program fragment

```
source : exp -> stm =
  \!x. (stm/let (exp/op !x op/+ e5) (\!y.
      (stm/let (exp/op !y op/+ !y) (\!z.
        (stm/return !z))))).
```

Listing 3.3: CLF encoding of listing 3.2

```
1  %     Frame                          Dest1 Dest2
2  cont  (f/let  (\!x. <>))              w1   [w2]
3  cont  (f/op1 op/+ <>)                 t2   [t1]
4  cont  (f/op2  (<> e10)  op/+)         t3    t1
5  cont  (f/let  (\!y. <>))              t1   [w2]
6  cont  (f/op1 op/+ <>)                 t5   [t4]
7  cont  (f/op2  (<> e15)  op/+)         t6    t4
8  cont  (f/let  (\!z. <>))              t4   [w2]
```

Listing 3.4: Continuations from listing 3.3

```
Location      Operands
?/x      <-   w1
?        <-   t2
t1       <-   val-10 + t3
?/y      <-   t1
?        <-   t5
t4       <-   val-15 + t6
?/z      <-   t4
```

Listing 3.5: Assembly for listing 3.4

distribute this program fragment to users who may run it after supplying the definition of `x`. The following is one such completion of the program in the form of a Celf query that interprets the program:

```
#query * 1 * 1
Pi w. eval (vbl/s (stm/let e10 source)) w -o { ret V w }.
```

We modified the program text to complete it, and then invoked the interpreter using `eval`.

Suppose we add a restriction that the user must not modify the program text in any way. This restriction is typical of program fragments distributed as compiled code. The following Celf query meets the additional design constraint using a latent rule.

```
#query * 1 * 1
Pi w1. Pi w2.
ret (val/e e10) w1 * cont (f/let source) w1 w2 -o { ret V w2 }.
```

The latent rule provides a cleared separation between the value to be provided at runtime and the program text available ahead of time. This gives us some insight into the nature of the static and dynamic content present in a SSOS specification, and moves us towards a binding-time analysis.

Observation 3.1. *`ret` relates runtime values to the destinations where a program may expect them. They likely contain the dynamic information that is required to run the Virtual Machine. We can exempt these terms from having to be of BSV. Continuations let us store the program text in a latent form that can be activated in a suitable environment. They likely contain the static information that is required to compile the program, and need to be of BSV.*

How exactly are continuations related to compiled code? We answer this question by examining the trigger rules. In each case, the continuation and frame play a central role in identifying which rule can fire. This observation roughly corresponds to the claim in thesis statement 1 that latent resources correspond to instructions in a virtual instruction set.

Listing 3.4 and Listing 3.5 attempt to illustrate this analogy. Listing 3.4 lists all the continuations that were triggered in a Celf execution of listing 3.3, in the order that they were triggered.

The destinations embedded in the continuations are all fresh variables generated by Celf when processing existential quantifiers.

Listing 3.5 informally transliterates the continuations into a three-operand pseudo assembly syntax. There are many parts of a frame that are not relevant to the mapping, so they have been replaced in the listing with <> for brevity. The destinations, since they have not yet been given a concrete representation, resemble temporary variables in the pseudo assembly that could be resolved to storage such as registers, memory locations. The destinations in the column `Dest1` relate continuations to resources on which they depend. These destinations appear as variables in the operand of the corresponding instruction. The destinations in column `Dest2` correspond to the ultimate destination to which the continuation must evaluate its result. Some of these destinations appear as the target memory locations of the corresponding instruction. However, these concepts do not exactly match, and consequently the columns are not identical.

The illustrations point to a few problems with applying the instruction set analogy to the current $\mathcal{L}$specification.

- We have not explained the meaning of `eval` in the static/dynamic dichotomy hypothesized so far, and the continuations listed in listing 3.4 do not account for the transitions caused by evaluation rules. Consequently, the pseudo assembly refers to undefined variables (`t2`, `t3`, `t5` and `t6`).
- Listing 3.5 shows that the "instructions" mention runtime values as constants (10 and 15). This means that the specification does not adequately separate static and dynamic content into latent and passive resources respectively.
- We have yet to explain how these "instructions" are generated without running the program, and how the generated instructions can be executed.

The following sections address these issues by incrementally revising the language specification to conform to S3OS.

3.1.1 Compile-time Computations and `eval`

Let us examine the evaluation rules to better understand the place of evaluations in the static/dynamic dichotomy and the instruction set analogy.

- `ev/let` and `ev/op` both peel apart expressions to produce continuations, but do not directly affect returned values. In the context of interpretation, these rules serve the purpose of traversing expressions independently of the dynamic environment. In the context of compilation, they apparently correspond to code generation, based on our previously stated desire to identify continuations as static and returned values as dynamic.
- `ev/return` also traverses expressions. Though it interacts with neither continuations nor returned values, its behavior is consistent with the interpretation of evaluations as code generating computations.
- `ev/num` interacts with dynamic content because it returns a value. This rule clashes with any interpretation of evaluation as code generation. In fact, the evaluation in this rule behaves much like a continuation – it contains a syntactic term and produces a runtime value when it fires.

```
ev/num :
  eval (vbl/e (exp/num N)) W




  -o { ret (val/e (exp/num N)) W }.
```

Listing 3.6: `exp/num` without immediates

```
f/num      : bin -> frame.
val/dummy : value.

ev/num :
  eval (vbl/e (exp/num N)) W
  -o { Exists x. ret val/dummy x *
       cont (f/num N) x W }.

tr/num :
  ret val/dummy X * cont (f/num N) X W
  -o { ret (val/e (exp/num N)) W }.
```

Listing 3.7: `exp/num` with immediates

```
1   %     Frame                    Dest1  Dest2
2   cont (f/let (\!x. <>))          w1    [w2]
3   cont (f/num !b10)              [t2]    t1
4   cont (f/op1 !op/+ <>)           t1    [t0]
5   cont (f/num !b5)               [t4]    t3
6   cont (f/op2 !(<> e10) !op/+)    t3     t0
7   cont (f/let !(\!y. <>))         t0    [w2]
8   cont (f/num !b15)              [t7]    t6
9   cont (f/op1 !op/+ <>)           t6    [t5]
10  cont (f/num !b15)              [t9]    t8
11  cont (f/op2 (<> e15) !op/+)     t8     t5
12  cont (f/let !(\!z. <>))         t5    [w2]
13  cont (f/num !b14)              [t10]   w2
```

Listing 3.8: Continuations from listing 3.3 with `f/num`

```
Location      Operands
?/x     <-    w1
t1      <-    val-10
?       <-    t1
t3      <-    val-5
t0      <-    val-10 + t3
?/y     <-    t0
t6      <-    val-15
?       <-    t6
t8      <-    val-15
t5      <-    val-15 + t8
?/z     <-    t5
w2      <-    val-14
```

Listing 3.9: Assembly for listing 3.8

The similarity between evaluations and continuations can be understood as follows:

Observation 3.2. *An* active *resource (as explained in 2.1.2) is a special case of latent resource with zero dependencies. Therefore, an evaluation can be transliterated into a continuation.*

Listing 3.7 illustrates the transformation by expanding `ev/num`. The expression is first moved into a continuation, and `tr/num` activates the continuation. We artificially introduce a dependency to preserve the style of continuations as resources that do not fire asynchronously. The dependency is a dummy value which does not exist in $\mathcal{L}$ and can be safely predicted at compile-time. Since this transformation is available, we can state the following as a requirement of S3OS.

S3OS requirement 3.1. *Evaluation rules must use continuations as intermediaries to return values that occur in the program text.*

Now our understanding of evaluations can be simplified as follows:

Observation 3.3. *The evaluation rules generate "instructions", which means they encode some form of compile-time computation for the purpose of compiling with SSOS. Therefore they have to be of BSV also.*

Based on this interpretation, evaluations will likely play little or no part in the virtual machine.

```
f/op2 : value -> op -> frame.

tr/op1 :
ret V1 X * cont (f/op1 B E2) X W
-o {
     Exists x. eval E2 x *
     cont (f/op2 V1 B) x W }.

tr/op2/add :

ret (val/e (exp/num N2)) X *
cont (f/op2 (val/e (exp/num N1)) op/+) X W
* !bin/add N1 N2 b0 N3 _
-o { ret (val/e (exp/num N3)) W }.
```

Listing 3.10: `f/op2` before separation

```
cont1 : dest -> frame -> dest
        -> dest -> type.
#mode cont1 - - - -.

f/op2 : op -> frame.

tr/op1 :
ret V1 X * cont (f/op1 B E2) X W
-o { Exists x1. ret V1 x1 *
     Exists x2. eval E2 x2 *
     cont1 x1 (f/op2 B) x2 W }.

tr/op2/add :
ret (val/e (exp/num N1)) X1 *
ret (val/e (exp/num N2)) X2 *
cont1 X1 (f/op2 op/+) X2 W *
!bin/add N1 N2 b0 N3 _
-o { ret (val/e (exp/num N3)) W }.
```

Listing 3.11: `f/op2` after separation

Listing 3.8 lists the continuations generated by the revised specification. Listing 3.9 introduces instructions that load immediate values in correspondence with `f/num`. These new instructions define the contents of variables that were previously being used without definition in listing 3.5. There are no remaining uses of undefined variables. However, there are destinations in the column `Dest1` enclosed in brackets to indicate that they do not appear as operands in the pseudo assembly. We will address this discrepancy later in the paper.

3.1.2 Separation using Destination Passing Style

So far, we have suggested that continuations should be static, but also noted that the $\mathcal{L}$ specification as it stood permitted runtime values to be stored in continuations. We need to enforce bounded static variation of continuations as a part of the style of specification. Section 3.1.1 also suggested that evaluations should be static, therefore it must also be subject to BSV. Therefore, we now incrementally revise the specification to indirectly pass dynamic content to evaluations and continuations using destination passing style.

S3OS requirement 3.2. *Returned values contain destinations and values, and values in turn contain object language terms. No rule should rewrite any of these terms from a returned value in the antecedent into a continuation or evaluation in the consequent.*

The first instance of a rule violating this requirement is `f/op2`. It takes the result of evaluating the first operand of a binary expression and places this runtime value inside a continuation. Listing 3.11 solves this problem by moving the value out of the continuation, and into a separate `ret` that can be tracked by its destination. This technique leads to the following observation:

Observation 3.4. *Any resource that contains both static and dynamic terms can be split into two resources that are related by a destination – one containing only static terms, and the other*

```
stm/let : exp -> (exp -> stm) -> stm.
f/let   : (exp -> stm) -> frame.

ev/let :
eval (vbl/s (stm/let E S)) W
-o { Exists y. eval (vbl/e E) y *
     cont (f/let S) y W }.

tr/let :
ret (val/e !V) X *
cont (f/let (\!e. S !e)) X W
-o {
     eval (vbl/s (S !V)) W }.
```

Listing 3.12: `tr/let` before separation

```
stm/let : exp -> (dest -> stm) -> stm.
f/let   : (dest -> stm) -> frame.

ev/let :
eval (vbl/s (stm/let E S)) W
-o { Exists y. eval (vbl/e E) y *
     cont (f/let S) y W }.

tr/let :
ret (val/e V) X *
cont (f/let (\!d. S !d)) X W
-o { Exists x. !ret (val/e V) x *
     eval (vbl/s (S !x)) W }.
```

Listing 3.13: `tr/let` after separation

```
exp/dest : dest -> exp.
f/dest   : dest -> frame.

ev/dest : eval (vbl/e (exp/dest D)) W
          -o { Exists x. ret val/dummy x * cont (f/dest D) x W }.

tr/dest : ret val/dummy X * cont (f/dest D) X W * !ret V D
          -o { ret V W }.
```

Listing 3.14: Destinations in expressions

```
source : dest -> stm =
  \!x. (stm/let (exp/op (exp/dest !x) op/+ e5) (\!y.
         (stm/let (exp/op (exp/dest !y) op/+ (exp/dest !y)) (\!z.
           (stm/return (exp/dest !z)))))).
```

Listing 3.15: Listing 3.3 with destinations

containing only dynamic terms.

`tr/let` is another instance of a rule requiring separation. The runtime value to be bound in the subsequent statement is substituted into a valuable term. Listing 3.13 again introduces a layer of indirection using destination passing to avoid embedding dynamic values in valuable terms. The value is placed in a `ret`, and the destination is substituted into the term to be evaluated.

Unlike in the previous cases, we need to also modify the syntax of $\mathcal{L}$ so that we can embed destinations inside expressions. We now need to provide semantics for handling destinations in expressions. Listing 3.14 specifies the semantics and listing 3.15 illustrates the changes to the encoding of listing 3.3.

The indirection through destination passing is not always the solution. For instance, if the dynamic term being rewritten into a continuation happens to be a destination, we would have to intervene in other ways. For this simple language however, the technique is sufficient to give a satisfactory separation of static and dynamic content occurring in evaluations and frames.

Listing 3.16 contains the continuations generated by the revised specification. Listing 3.17

```
1  %     Dest0 Frame                 Dest1  Dest2
2  cont        (f/let (\!x. <>))      w1    [w2]
3  cont        (f/dest t13)          [t3]    t2
4  cont        (f/op1 <>)             t2    [t1]
5  cont        (f/num b5)            [t4]    t5
6  cont1  t0   (f/op2 op/+)           t5     t1
7  cont        (f/let (\!y. <>))      t1    [w2]
8  cont        (f/dest t14           [t8]    t7
9  cont        (f/op1 <>)             t7    [t6]
10 cont        (f/dest t14)          [t9]    t12
11 cont1 t10   (f/op2 op/+)           t12    t6
12 cont        (f/let (\!z. <>))      t6    [w2]
13 cont        (f/dest t15)          [t11]   w2
```

Listing 3.16: Separated continuations from listing 3.15

```
Location      Operands
[t13]/x  <-   w1;
 t2      <-   t13;
[t0]     <-   t2;
 t5      <-   val-5;
 t1      <-   t0 + t5;
[t14]/y  <-   t1;
 t7      <-   t14;
[t10]    <-   t7;
 t12     <-   t14;
 t6      <-   t10 + t12;
[t15]/z  <-   t6;
 w2      <-   t15;
```

Listing 3.17: Assembly for listing 3.16

resembles a typical instruction set more closely than listing 3.9. We now know all the variables to which values get assigned. Previously we could not identify all these variables because runtime values got moved back into continuations. However, we had to consult the full Celf trace to reconstruct some of the variables occurring in the pseudo assembly – the destinations do not always occur in the column `Dest2`. This shortcoming will be resolved in the next section.

Now we know that valuable terms and frames only mention terms that occur in the program source and destinations generated by the $\mathcal{L}$ specification. Once we ensure that the destinations and valuable terms in the specification satisfy BSV, we can also prove that continuations and evaluations satisfy BSV. The BSV property for destinations is addressed in the next section. Ideally, the BSV property for valuable terms would arise from compositionality or semicompositionality of the specification. However, since we use substitutions, we state the following requirement instead:

S3OS requirement 3.3. *Every forward inference rule in an S3OS specification produces valuable terms that are smaller than those occurring in the antecedent.*

After separating the dynamic content out of `tr/let`, the $\mathcal{L}$ specification does meet the requirement. Substitutions in general allow a term to grow and have a dynamic size. However, destinations in S3OS do not contain redexes and can be safely substituted into valuable terms.

Note that the requirement prevents the encoding of repeated computation such as loops and function applications using evaluations, returns and continuations. Chapter 4 revises S3OS to overcome this limitation.

3.2 Compilation

This section explains how to synthesize a compiler from a specification using linear logical approximations, and substantiates thesis statement 2. The basic idea is to make all continuations persistent using the `!` modality and run the specification to *saturation*. This change allows us to collect all the resources generated while executing the language specification.

The concept of saturation was developed in the context of Datalog and deductive databases [GR68, Min96], where forward inference was used to populate a database with facts. Our understanding of saturation is also influenced by the work of McAllester and Ganzinger on logical algorithms [GM02]. In Celf, the persistent context is analogous to the database. Forward inference was allowed to match facts already present in the database and add new facts to the database. The logic program as a whole is executed until no new facts can be generated by any of the inference rules, at which point the database is said to be saturated and the program terminates.

In order to establish that a specification can saturate, we need to know that it generates a finite number of continuations. To this end, we have already worked towards putting the specification in a form in which continuations satisfy BSV. We also need to know that execution of the specification cannot cycle between a finite set of states and prevent termination. We can produce cycles by applying inference rules which repeatedly produce and consume the same linear resources, thus preventing *quiescence*. We make all resources persistent to avoid cycling and solve a few other problems. The operational justification for adding persistence follows:

Evaluations should be persistent because they indicate when a compile-time computation has occurred and allows us to not repeat them. However, rules such as `ev/op` prevent saturation because the forward chaining interpretation of the existential quantifier generates a fresh variable in the persistent context for each application of the rule. This problem is related to the previously stated problem of establishing the BSV property for destinations. Section 3.2.1 describes a technique called *Skolemization with equality* that is used to suppress the ability of existential quantifiers to produce an arbitrary number of fresh symbols and satisfy BSV.

We also need to collect `ret` resources in some form, because we need to know whether *any* value may be returned at a specific destination and activate a continuation waiting at that destination. However, the collection process should not depend on the dynamic content currently embedded in returned values. Section 3.2.2 explains how to use *mode-driven erasure* to eliminate all occurrences of values in the specification while still permitting the collection of all evaluations and continuations.

3.2.1 Skolemization with Equality

The ability of the existential quantifier to generate fresh symbols can be restricted by equating the quantified variable to some Skolem function. Simmons and Pfenning use this technique in [SP09] to approximate the execution of programs and derive program flow analyses. Listing 3.18 illustrates the style of constraints used in [SP09] on `ev/let`. The definition of the source language is modified to annotate every source term with a *label*. Next, the existentially quantified variable is tied to the label of the expression in the antecedent. The equality is considered persistent, so the variable generated by the quantifier will always be represented by the same label. Since every firing of the rule produces a variable with the same persistent representation, the existentially quantifier no longer poses a problem for saturation.

The code in listing 3.18 is not valid CLF syntax. We need to substitute the Skolem function for every occurrence of the existentially quantified variable.

For the purpose of compilation, we do not wish to modify the object language definition. Listing 3.19 illustrates a new skolemization strategy that can be used without modifying the

```
label       : type.
dest/label : label -> dest.
stm/let     : label -> exp -> (dest -> stm) -> stm.

ev/let : eval (vbl/s (stm/let L E S)) W
         -o { Exists x = (dest/label L).
              eval (vbl/e E) x * cont (f/let S) x W }.
```

Listing 3.18: Skolemization with Equality

```
d/6      : dest -> dest.
d/7      : dest -> dest.

ev/let : eval (vbl/s (stm/let E S)) W
         -o { eval (vbl/e E) (d/6 W) *          % Exists x = (d/6 W)
              cont (f/let S) (d/6 W) W }.

tr/let : ret V X * cont (f/let (\!d. S !d)) X W
         -o { !ret V (d/7 W) *                  % Exist x = (d/7 W)
              eval (vbl/s (S !(d/7 W))) W }.
```

Listing 3.19: `let` rules with constructed destinations

object language definition. Each quantified destination is substituted with a new destination generated by wrapping the *target destination* with a constructor (for example `d/6`). The target destination is the destination at which the computation in the antecedent of a forward inference rule must ultimately return. Due to the syntactic conventions used in the specification, the target destination is always called `W` in each rule. Each destination constructor is only used in one part of the $\mathcal{L}$ specification. Since there are finitely many rules and occurrences of existential quantifiers in them, we only need a finite number of destination constructors such as `d/6` and `d/7`.

This skolemization strategy outlined so far is too crude of an approximation to produce useful compiled code. Consider what happens when we replace the variables in listing 3.16 with the skolemized destinations.

```
t1  = t6  = (d/6 w2)
t2  = t7  = (d/2 (d/6 w2))
t0  = t10 = (d/3 (d/6 w2))
t5  = t12 = (d/4 (d/6 w2))
t13 = t14 = t15 = (d/7 w2)
t3  = t8  = t9  = t11 = (d/5 (d/2 (d/6 w2)))
```

Symbols that were intended to be distinct are given the same representation. The destinations on which continuations should wait are approximated, and consequently the set of returned values which may trigger a continuation are also approximated. We could precisely determine which returned values should trigger a continuation prior to skolemization because we implicitly observed the following requirement:

S3OS requirement 3.4. *Continuations always wait on existentially quantified destinations, and*

no two continuations can wait on destinations bound at the same place. This ensures that any returned value can activate at most one continuation.

There is another problem. We substitute destinations into let statements. Before skolemization, $\mathcal{L}$ generated a fresh destination for each substitution. Since these destinations (`t13`, `t14` and `t15` in the example) are equivalent after skolemization, frames and expressions that were previously distinct are now equivalent.

The shared destinations arise after skolemization because the `ev/let` and `tr/let` rules each create a new evaluation or continuation at the same target destination `W` as the antecedent. If there are ever two let bound statements in a program which are syntactically equivalent but do not have the same subterm occurrence in the program, the semantics will place the continuations generated for each of these statements at destinations constructed by applying the same constructor multiple times to the same target destination!

We should strive to construct continuations for any *positionally non-equivalent* source terms at distinct destinations

Definition 3.1. *Two source terms are considered* positionally equivalent *in the program text if and only if they have the same occurrence in the CLF encoding of a program.*

Suppose there are two positionally equivalent terms of the form `\x : dest. E x`*. If we substituted the same destination for* `x` *in positionally equivalent occurrences of* `E`*, the results are also considered positionally equivalent.*

To give the appropriate treatment to positionally nonequivalent terms, we need to classify frames as terminals or non-terminals.

Definition 3.2. *A frame* `F` *can be classified as* terminal *if and only if there exists no forward inference rule which matches* `cont F X W` *in the antecedent and produces a new continuation or evaluation.*

The terminals for $\mathcal{L}$ are `f/num`, `f/op2` and `f/dest`. `f/let` and `f/op1` are nonterminals because `tr/let` and `tr/op1` match these frames and produce new continuations or evaluations. Continuations that contain nonterminal frames are classified as nonterminal resources. Evaluations are also classified as nonterminal resources because evaluation rules always produce new continuations and evaluations as explained in section 3.1.1. Together, all the nonterminal resources and the rules matching them are responsible for compile-time computation.

The skolemization strategy requires that nonterminal resources should be produced only at freshly created destinations, so that destinations further derived by wrapping additional constructors are also guaranteed to be fresh. Terminals may be produced by reusing the target destination `W` from the antecedent of a rule. Listing 3.21 transforms `ev/let` to meet the new constraint. We introduce a new terminal frame `f/dummy` which is used to copy the result computed at a fresh destination to the target destination. The rule for `f/dummy` is given in listing 3.19. The illustrated transformation is called freshening of nonterminals. Since this transformation is available, we can formally add another restriction to S3OS:

S3OS requirement 3.5. *Any forward inference rule matching a nonterminal in the antecedent at target destination* `W` *is not allowed to produce another nonterminal at* `W`*.*

```
%before skolemization
ev/let :
  eval (vbl/s (stm/let E S)) W
  -o { Exists x. eval (vbl/e E) x *
       cont (f/let S) x W }.
%

%after skolemization
ev/let :
  eval (vbl/s (stm/let E S)) W
  -o { eval (vbl/e E) (d/6 W) *
       cont (f/let S) (d/6 W) W }.
%
```

Listing 3.20: `ev/let` before freshening

```
%before skolemization
ev/let :
  eval (vbl/s (stm/let E S)) W
  -o { Exists x. eval (vbl/e E) x *
       Exists y. cont (f/let S) x y *
       cont f/dummy y W }.

%after skolemization
ev/let :
  eval (vbl/s (stm/let E S)) W
  -o { eval (vbl/e E) (d/6 W) *
       cont (f/let S) (d/6 W) (d/8 W) *
       cont f/dummy (d/8 W) W }.
```

Listing 3.21: `ev/let` after freshening

```
f/dummy : frame.

tr/dummy : !ret V X * !cont f/dummy X W
          -o { !ret V W }.
```

Listing 3.22: `tr/dummy`

This style of S3OS specification in which existentially quantified variables are realized as constructed destinations is called S3OS-sko. Appendix B.2 includes S3OS-sko for the entire language. Note that evaluations, returns and continuations have not yet been made persistent in S3OS-sko. We need to establish that S3OS and S3OS-sko are somehow equivalent in their behavior. The following definitions and property do so:

Definition 3.3. *Suppose that:*

- Ω *is a set of destinations generated by quantifiers.*
- Γ *is a state containing S3OS resources that may mention destinations in* Ω*.*

A S3OS-sko state Δ *is said to be a* realization *of* (Γ, Ω) *if* $\Delta = \sigma\Gamma$*, where* σ *is a substitution of destinations constructed using constructors from the S3OS-sko signature for variables in* Ω*.*

Theorem 3.1. *A S3OS specification can produce a trace* Θ *when running an input program if and only if the corresponding S3OS-sko specification can produce a trace* Θ' *such that for all* n*, the* n^{th} *state of* Θ' *is a realization of the* n^{th} *state of* Θ*.*

An informal proof of this property is given in appendix A.1.1, with necessary lemmas.

We have now demonstrated how to synthesize an equivalent language specification that does not use unconstrained existential quantifiers to generate destinations. Consequently, all resources can be made persistent without worrying about an infinite number of arbitrary variables created by existential quantifiers.

The S3OS-sko $\mathcal{L}$ can be seen in action in listing 3.23, which lists all the continuations accumulated for the sample program fragment[3]. With the exception of the dummy continuations at

[3]We would accumulate a large number of dummy continuations if we freshened the target destinations of all nonterminals, so we optimized a few cases to produce readable traces. The optimization is only applicable under

```
1  % Dest0   Frame                      Dest1      Dest2  | Location     Operands
2            (f/let <>)                   w1         w2   |    7-w2 <-        w1;
3            (f/dest 7-w2)      [5-2-6-w2]    2-6-w2   |  2-6-w2 <-      7-w2;
4            (f/op1 <>)              2-6-w2     [6-w2]  |  3-6-w2 <-    2-6-w2;
5            (f/num !b5)        [1-4-6-w2]    4-6-w2   |  4-6-w2 <-     val-5;
6    3-6-w2  (f/op2 !op/+)           4-6-w2       6-w2  |    6-w2 <-    3-6-w2 + 4-6-w2;
7            (f/let <>)                6-w2     [8-w2]  |  7-8-w2 <-      6-w2;
8            (f/dest 7-8-w2)    [5-<>-w2]  2-6-8-w2   |2-6-8-w2 <-    7-8-w2;
9            (f/op1 <>)            2-6-8-w2   [6-8-w2]  |3-6-8-w2 <-  2-6-8-w2;
10           (f/dest 7-8-w2)    [5-<>-w2]  4-6-8-w2   |4-6-8-w2 <-    7-8-w2;
11 3-6-8-w2  (f/op2 !op/+)         4-6-8-w2     6-8-w2  |  6-8-w2 <-  3-6-8-w2 + 4-6-8-w2
12           (f/let <>)              6-8-w2   [8-8-w2]  |7-8-8-w2 <-    6-8-w2;
13           (f/dest 7-8-8-w2)   [5-<>w2]     8-8-w2   |  8-8-w2 <-  7-8-8-w2;
14           (f/dummy)               8-8-w2       8-w2  |    8-w2 <-    8-8-w2;
15           (f/dummy)                 8-w2         w2  |      w2 <-      8-w2;
```

Listing 3.23: Continuations from listing 3.16 after skolemization and freshening

Listing 3.24: Assembly

the end, the continuations are unchanged. No two continuations have the same destination in the `Dest0` and `Dest1` columns, same as before skolemization.

Listing 3.24 translates the continuations to instructions. The dummy continuations translate to straightforward copy instructions. There is one important improvement in this translation compared to the translation in listing 3.17 – we no longer need to consult the full execution trace of the program to reconstruct the memory locations. Instead, we only need to know the rule matching the continuation and the target destination. For instance, consider the continuation on line 7. The frame is `f/let`, and the only matching rule is `tr/let` (see listing 3.21). We also know from the column `Dest2` that the target destination `W` is `(d/8 w2)`. The rule returns the value to `(d/7 W)`, i.e. `(d/7 (d/8 w2))`.

The S3OS-sko specification brings us one step closer to compilation through saturation.

3.2.2 Mode-driven Erasure

A S3OS-sko specification in which all evaluations, continuations and returns have been made persistent cannot yet be run as a compiler. The trigger rules in the specification should match nonterminal continuations and generate the continuations in the consequent if they have not already been produced. In order to activate all pertinent nonterminal continuations, we need to know where values may be returned, but we should not depend on the value itself. Intuitively, we wish to have `ret : dest -> type` instead of `ret : value -> dest -> type`. This change in the signature would require us to modify every occurrence of `ret` in the forward inference rules to remove the value.

The rules obtained by mechanically erasing every value are not necessarily well-moded. Consider what happens to `tr/op2/add` after erasing all values from `ret` resources (see listing 3.1 for mode declaration of `bin/add`). The input variables `N1` and `N2` are no longer ground, as can

very specific circumstances, We leave a more complete description of optimizations to future work.

```
tr/op1 :
!ret X * !cont (f/op1 B E2) X W
-o { !ret (d/3 W) * !eval E2 (d/4 W) *
     !cont1 (d/3 W) (f/op2 B)
            (d/4 W) W }.

tr/op2/add :
!ret X1 * !ret X2 *
!cont1 X1 (f/op2 op/+) X2 W *
!bin/add N1 N2 b0 N3 _
-o { !ret W }.
```

Listing 3.25: After partial erasure

```
tr/op1 :
!ret X * !cont (f/op1 B E2) X W
-o { !ret (d/3 W) * !eval E2 (d/4 W) *
     !cont1 (d/3 W) (f/op2 B)
            (d/4 W) W }.

tr/op2/add :
!ret X1 * !ret X2 *
!cont1 X1 (f/op2 op/+) X2 W

-o { !ret W }.
```

Listing 3.26: After mode-driven erasure

be seen in listing 3.25. However, due to separation, the result `N3` is no longer referenced. Therefore the need to compute `N3` is obviated, as illustrated in listing 3.26. The technique of repeatedly removing any computation that is not well-moded from the specification is called mode-driven erasure.

All the rules for compiling binary operations are listed in listing 3.26. `ev/op` is unchanged because it was originally describing a purely compile-time computation. `tr/op1` also largely preserves the structure of the original rule – the evaluation of the second operand must begin after the evaluation of the first operand is completed. However, the computed value of the first operand is never referenced. `tr/op2` states that when both operands become available, the result of the addition will be returned at the correct destination *somehow*. The underlying specification of addition is not referenced because that computation has been deferred to runtime.

To ensure that mode-driven erasure can never result in the erasure of an evaluation or a continuation, we need to refine a S3OS requirement. Previously we had stated that no forward inference rule should copy a returned value from the antecedent into a continuation or evaluation in the consequent. Instead:

S3OS requirement 3.6. *A value occurring in a forward inference rule is considered to be dynamic content if it (a) occurs in a returned value in the antecedent, (b) is computed in the antecedent by a predicate which takes returned values as inputs, or (c) is in the transitive closure of all values computed by predicates which take inputs from (a) and (b). None of these dynamic terms should be rewritten into a continuation or evaluation in the consequent.*

$\mathcal{L}$ as specified at the end of section 3.1.2 already meets this requirement.

The illustrated style of specification is called S3OS-comp. Appendix B.3 includes the S3OS-comp for all of $\mathcal{L}$. The following theorems establish the validity of compiling with S3OS-comp:

Theorem 3.2. *After mode-driven erasure, an S3OS-comp specification saturates.*

An informal proof is given in appendix A.1.2.

Theorem 3.3. *A S3OS-comp specification is guaranteed to collect a set of continuations that is a superset of that which could have been collected by S3OS-sko.*

An informal proof is given in appendix A.1.3.

To conclude this section, let us review how to start compilation.

```
#query * 1 * 1
Pi w1. Pi w2. !ret w1 * !cont (f/let source) w1 w2 -o { !ret w2 }.
```

The query compiles the source term in `source` into the required collection of continuations. The query also serves as a specification of how one may execute a compiled program. In this case, a value must be return at the predetermined destination `w1`.

3.3 Virtual Machine

The continuations accumulated by the compiler described in the previous section may be in a form that is convenient for translation into a simple instruction set. However, a reference implementation of a runtime environment for the continuations would be useful. Furthermore, the collected continuations still mention program text that should ideally be unnecessary. To substantiate thesis statement 3, we synthesize a virtual machine for the continuations by applying the techniques described during compilation – persistence, skolemization and mode-driven erasure.

Let us start with the S3OS-sko specification with one modification – all continuations are made persistent using the `!` modality. Evaluations and returned values are still linear. The basic idea is to synthesize a specification that is compatible with the persistent continuations collected by the S3OS-comp specification. We call the resulting specification a S3OS-JIT specification because it resembles a just-in-time compiler compatible with the S3OS-comp specification. By just-in-time, we mean that the program is being compiled while it is being run with all its inputs[4].

Changes in persistence are generally not modular, so we need to establish that the specification still computes the same results. In general, introducing persistence can potentially change:

- the returned values that may match a continuation by leaving around more candidate continuations.
- the order in which trigger rules may fire because a rule matched a persistent resource instead of waiting for a linear instance to be created.
- the number of times a continuation can be activated, because it did not get consumed when matched and is left around for other matchings.

We have observed certain guidelines (either explicitly or implicitly) during the design of the S3OS specification for $\mathcal{L}$ that prevent the distortions mentioned above. For instance, due to requirement 3.4, we know that no two continuations wait on the same destination. Please refer to lemma A.1.3 for a proof of the same property in S3OS-sko. Therefore, a value returning at a particular destination can unambiguously identify the continuation to be triggered.

We have not changed the order in which trigger rules may fire, because we have implicitly conformed to the following requirement:

S3OS requirement 3.7. *Continuations must always be produced before, or at the same time as the computations which may trigger them.*

[4]We say S3OS-JIT purely for the sake of setting up the analogy. A compiler which generates each instruction right before it is required is not particularly practical.

This requirement makes the timing of a trigger rule depend solely on the availability of the returned value. It allows us to create the continuation earlier, which is essential if we are to execute code compiled ahead-of-time (i.e without running the program on inputs as in the case of S3OS-comp).

In order to not change the number of times a continuation can be activated, we need the following requirements:

S3OS requirement 3.8. *A rule may not produce more than one set of computations that may trigger a continuation.*

If multiple values were allowed to return to one destination where one ephemeral continuation waited, one of the values would have nondeterministically triggered the continuation. When such a continuation is made persistent, all returned values would be processed. The following is also required:

S3OS requirement 3.9. *The set of returned values triggering a continuation cannot entirely be persistent, because persistent resources can be instantiated an arbitrary number of times.*

This intuition is more formally represent by the following definition and property:

Definition 3.4. *A S3OS-JIT state Δ is called a* compiling instantiation *of a S3OS-sko Γ if and only if:*

- Δ *contains the same multiset of evaluations as* Γ
- Δ *contains the same multiset of returns as* Γ
- Δ *contains a superset of the continuations in* Γ *made persistent.*

Theorem 3.4. *A S3OS-sko specification can produce a trace Θ when evaluating a program if and only if the corresponding S3OS-JIT specification can produce a trace Θ' such that for all n, the nth state of Θ' is a compiling instantiation of the nth state of Θ.*

An informal bisimulation proof is given in appendix A.1.4.

Listing 3.27 illustrates the evaluation rules in the S3OS-JIT specification of $\mathcal{L}$, which we shall use as examples for the next few transformations. To synthesize a virtual machine, we eliminate continuations from the consequents of all forward inference rules. Instead we expect that the CLF context is loaded with all the continuations generated by the ahead-of-time compiler (listing 3.28).

The resulting virtual machine is not particularly efficient because it still refers to source expressions in the continuations and evaluations. We can use a technique that is closely related to mode-driven erasure to remove source expressions from the VM. Intuitively, we wish to have `eval : dest -> type` instead of `eval : value -> dest -> type`.

This strategy leads to ambiguity in evaluation rules. After erasure, the rules match any evaluation (listing 3.29), and the VM will nondeterministically pick the destination at which to return a dummy value or start the next evaluation. However, the information required for disambiguation can be found in the compiled continuations. In any rule that formerly produced a continuation at the target destination, we can match on those same continuations in the antecedent. Therefore, we take the continuations in listing 3.27 and match them in the antecedents of rules in listing

```
eval (vbl/e (exp/num N)) W
-o { ret val/dummy (d/1 W) *
     !cont (f/num N) (d/1 W) W }.

eval (vbl/e (exp/dest D)) W
-o { ret val/dummy (d/5 W) *
     !cont (f/dest D) (d/5 W) W }.

eval (vbl/s (stm/let E S)) W
-o { eval (vbl/e E) (d/6 W) *
     !cont (f/let S) (d/6 W) (d/8 W) *
     !cont f/dummy (d/8 W) W }.
```

Listing 3.27: Persistent continuations

```
eval (vbl/e (exp/num N)) W
-o { ret val/dummy (d/1 W)
                           }.

eval (vbl/e (exp/dest D)) W
-o { ret val/dummy (d/5 W)
                           }.

eval (vbl/s (stm/let E S)) W
-o { eval (vbl/e E) (d/6 W)

                            }.
```

Listing 3.28: Stop producing continuations

```
ev/num : eval W

         -o { ret val/dummy (d/1 W) }.

ev/dest : eval W

          -o { ret val/dummy (d/5 W) }.

ev/let : eval W


         -o { eval (vbl/e E) (d/6 W) }.
```

Listing 3.29: Erase valuables

```
eval W *
!cont (f/num N) (d/1 W) W
-o { ret val/dummy (d/1 W) }.

eval W *
!cont (f/dest D) (d/5 W) W
-o { ret val/dummy (d/5 W) }.

eval W *
!cont (f/let S) (d/6 W) (d/8 W) *
!cont f/dummy (d/8 W) W
-o { eval (d/6 W) }.
```

Listing 3.30: Disambiguate by matching continuations

3.30.

The resulting specification is called a S3OS-VM specification. Its validity is stated as follows:

Definition 3.5. *A S3OS-VM state Δ is called an* ahead-of-time instantiation *of a S3OS-JIT state Γ if and only if:*

- *Δ contains the same multiset of evaluations as Γ, but with valuable terms erased.*
- *Δ contains the same multiset of returns as Γ*
- *Δ contains all the continuations that could possibly be generated by an S3OS-JIT specification when run on Γ, and making all the continuations persistent.*

Theorem 3.5. *A S3OS-JIT specification can produce a trace Θ when evaluating a program if and only if the corresponding S3OS-VM specification can produce a trace Θ' such that for all n, the nth state of Θ' is an ahead-of-time instantiation of the nth state of Θ.*

An informal bisimulation proof is given in appendix A.1.5. The proof requires an inversion lemma which is also stated in the appendix.
After erasure, evaluations behave like jump instructions found in popular instruction sets.

After the transformation, the continuations matched in the antecedent of trigger rules can

```
frame    : type.                          inst     : type.
f/op1    : op -> valuable -> frame.       i/op1    : inst.
f/op2    : op -> frame.                   i/op2    : op -> inst.
f/let    : (dest -> stm) -> frame.        i/let    : inst.

tr/frame: frame -> inst -> type. #mode tr/frame + -.
tr/frame/op1   : tr/frame (f/op1 B E) (i/op1).
tr/frame/op2   : tr/frame (f/op2 B)   (i/op2 B).
tr/frame/let   : tr/frame (f/let S)   (i/let).

cont   : frame -> dest -> dest -> type.          #mode cont  - - -.
comp   : inst  -> dest -> dest -> type.          #mode comp  - - -.
cont1  : dest -> frame -> dest -> dest -> type. #mode cont1 - - - -.
comp1  : dest -> inst  -> dest -> dest -> type. #mode comp1 - - - -.

tr/cont  : !cont F X W * tr/frame F I -o { !comp I X W }.
tr/cont1 : !cont1 X1 F X2 W * tr/frame F I -o { !comp1 X1 I X2 W }.
```

Listing 3.31: Code simplification

```
tr/op1 :
 ret V1 X * !cont  (f/op1 B E2) X W
 -o { ret V1 (d/3 W) * eval (d/4 W) }.
```

Listing 3.32: `tr/op1` with unused valuables

```
tr/op1 :
 ret V1 X * !comp i/op1 X W
 -o { ret V1 (d/3 W) * eval (d/4 W) }.
```

Listing 3.33: Simplified `tr/op1`

also be streamlined. For example, consider `tr/op2` reproduced in listing 3.32. `B` and `E2` are no longer referenced in the consequent of the rule, and can be removed. In general, source terms can be removed from frames in the virtual machine when:

- No resource in the consequent of the relevant trigger rules reference the source term. In $\mathcal{L}$, `f/num` and `f/dest` need their source terms in order to return the correct value.
- Ambiguity is not introduced into continuations. This is not a problem because we have already ensured that no two continuations can wait on the same destination. Therefore, we can distinguish between continuation in the compiled program even without all of the terms in the frame.
- No trigger rule matches on the contents of a source term to determine whether the rule can fire. In $\mathcal{L}$, we need to preserve the binary operator in `f/op2` because it determines whether `tr/op2/add` can fire.

We can easily simplify continuations and frames in CLF using a few more forward inference rules. Listing 3.31 gives a few representative cases. Frames are translated to terms of a new type `inst` and continuations are translated to a new type `comp` (for compiled code). Now we can use the streamlined `comp` instead of `cont` in the virtual machine, as illustrated in Listing 3.33.

Listing 3.34 and listing 3.35 reproduce the compiled continuations and pseudo assembly from listing 3.23 and listing 3.24 in light of the erasure of information from evaluations. Any previously unexplained destinations are now understood in light of the introduction of jump statements. Also note that we no longer use <> to abbreviate terms in frames that are not used

```
 1  % Dest0   Inst                     Dest1      Dest2
 2            (i/let)                     w1         w2
 3
 4
 5            (i/dest 7-w2)       5-2-6-w2    2-6-w2
 6            (i/op1)               2-6-w2      6-w2
 7
 8
 9            (i/num !b5)         1-4-6-w2    4-6-w2
10   3-6-w2   (i/op2 !op/+)         4-6-w2      6-w2
11            (i/let)                 6-w2      8-w2
12
13
14            (i/dest 7-8-w2)  5-2-6-8-w2 2-6-8-w2
15            (i/op1)             2-6-8-w2    6-8-w2
16
17
18            (i/dest 7-8-w2)  5-4-6-8-w2 4-6-8-w2
19 3-6-8-w2   (i/op2 !op/+)       4-6-8-w2    6-8-w2
20            (i/let)               6-8-w2    8-8-w2
21
22
23            (i/dest 7-8-8-w2) 5-8-8-w2    8-8-w2
24            (i/dummy)             8-8-w2      8-w2
25            (i/dummy)               8-w2        w2
```

Listing 3.34: Simplified continuations

```
 1   Location      Operands
 2       7-w2 <-          w1;
 3       jump     5-2-6-w2;
 4 target 5-2-6-w2:
 5     2-6-w2 <-        7-w2;
 6     3-6-w2 <-      2-6-w2;
 7       jump     1-4-6-w2;
 8 target 1-4-6-w2:
 9     4-6-w2 <-       val-5;
10       6-w2 <-      3-6-w2 + 4-6-w2
11     7-8-w2 <-        6-w2;
12       jump   5-2-6-8-w2;
13 target 5-2-6-8-w2:
14   2-6-8-w2 <-      7-8-w2;
15   3-6-8-w2 <-    2-6-8-w2;
16       jump   5-4-6-8-w2;
17 target 5-4-6-8-w2:
18   4-6-8-w2 <-      7-8-w2;
19     6-8-w2 <-    3-6-8-w2 + 4-6-8-
20   7-8-8-w2 <-      6-8-w2;
21       jump     5-8-8-w2;
22 target 5-8-8-w2:
23     8-8-w2 <-    7-8-8-w2;
24       8-w2 <-      8-8-w2;
25         w2 <-        8-w2;
```

Listing 3.35: Assembly

in the translation, because we have removed those unused terms from the definition of `inst`. Appendix B.4 includes the simplified S3OS-VM for all of $\mathcal{L}$.

3.4 Summary

We conclude this chapter with a summary of how we synthesize the specification of a compiler and virtual machine from SSOS, and then feed a program through these specifications.

Specification synthesis is outlined in figure 3.1. First, an SSOS specification is converted to an S3OS specification using the techniques discussed in section 3.1.1 and 3.1.2. We also make sure that the specification meets the structural requirements of skolemization by equality. Next, we apply skolemization by equality to the destinations in the S3OS specification as described in section 3.2.1, since we require concrete destinations for both the compiler and the VM.

The specification in skolemized destination form is then converted to a compiler using persistence and mode-driven erasure as described in section 3.2.2. Skolemized destination form is also converted to a Virtual Machine through the appropriate use of persistence and erasure as described in 3.3. Finally, information from the VM is used to simplify the compiled code and remove references to source terms from the compiled code and the VM. Appendix B gives the S3OS-sko, S3OS-comp and simplified S3OS-VM specification for all of $\mathcal{L}$.

Program manipulation is described in figure 3.2. The user would have to supply a parser for

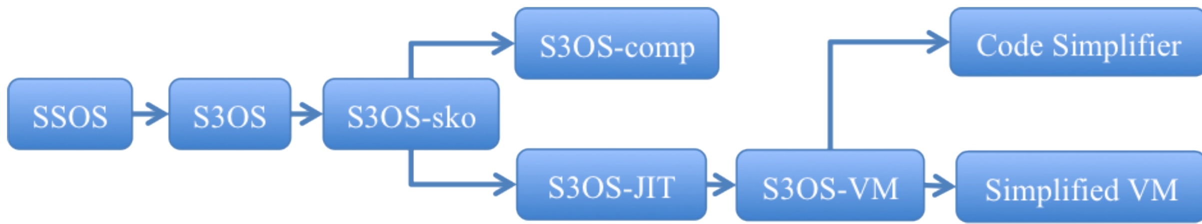

Figure 3.1: Specification synthesis

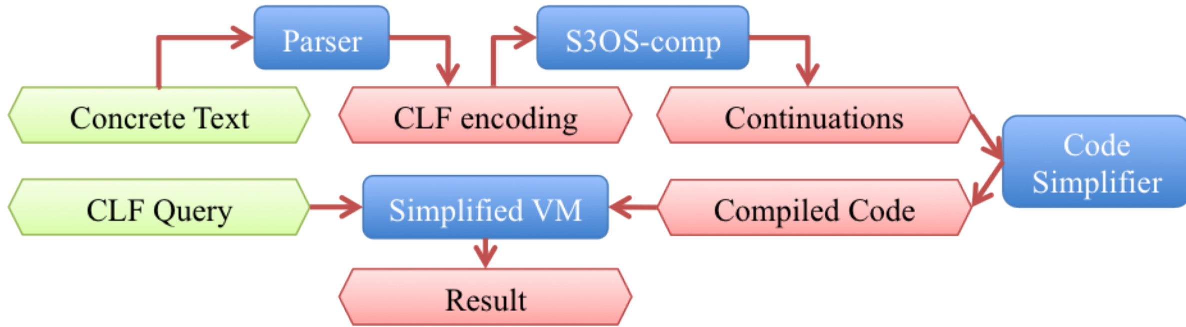

Figure 3.2: Program transformation

the object language or directly encode the program in CLF. A CLF encoding can then be passed to the synthesized compiler and run to saturation using the appropriate Celf query. The collected continuations are simplified using the synthesized code simplifier. The compiled product can be executed in the reference virtual machine if it is used to seed the CLF context of the VM and run using a CLF query that supplies all the necessary arguments.

Most of the specification synthesis is quite mechanical and easy to automate. The most complicated part is converting an SSOS specification to S3OS, because SSOS refers to a very broad style and S3OS is rather specific. Therefore, it may not be straightforward to automate in general. This subject is beyond the scope of this dissertation. However, it should be simpler to check whether a set of forward inference rules conform to the structural requirements of S3OS – they simply need to match the grammar in listing 3.36. This loosely defined grammar captures all of the structural requirements of compiler and VM synthesis that were previously stated informally:

- Evaluation rules may produce any evaluation or continuation. They should not return any value, except the dummy value.
- Continuations always wait on existentially quantified destinations, and no two continuations can wait on destinations bound at the same place. This in effect ensures that any returned value can activate at most one continuation.
- Any forward inference rule matching a nonterminal in the antecedent at target destination `W` is not allowed to produce another nonterminal at `W`.
- No rule may produce more than one resource at one destination.
- A returned value which may trigger a continuation cannot be persistent.
- Continuations must always be produced before, or at the same time as the computations

```
rule :: <evaluation> | <trigger>

evaluation :: eval <valuable> W
              -o { Exists x. <ev-cons(x)> * cont <terminal-frame> x W }.

ev-cons(W) :: eval <valuable> W
            | ret val/dummy W
            | Exists x. <ev-cons(x)> * cont <frame> x W    % x fresh

frame :: <terminal-frame> | <non-terminal-frame>

trigger :: ret <value> X * cont <non-terminal-frame> X W *
           <optional-backward-chaining-computations> -o { tr-cons(W) }.
         | ret <value> X1 * ret <value> X2 *
           <optional-backward-chaining-computations> *
           cont1 X1 <non-terminal-frame> X2 W -o { tr-cons(W) }.
         | ret <value> X * cont <terminal-frame> X W *
           <optional-backward-chaining-computations> -o { ret <value> W }.
         | ret <value> X1 * ret <value> X2 *
           <optional-backward-chaining-computations> *
           cont1 X1 <terminal-frame> X2 W -o { ret <value> W }.

tr-cons(W) :: Exists x. <tr-cons'(x)> * cont <terminal-frame> x W
            | Exists x1. ret <value> x1 * Exists x2. <tr-cons'(x2)> *
              cont1 x1 <terminal-frame> x2 W

tr-cons'(W) :: ret <value> W
             | Exists x. <tr-cons'(x)> * cont <frame> x W
             | Exists x1. ret <value> x1 * Exists x2. <tr-cons'(x2)> *
               cont1 x1 <frame> x2 W
```

Listing 3.36: S3OS requirement specification

which may trigger them.

The binding-time analysis guaranteeing separation of static and dynamic terms must be performed separately, and may be possible to automate only in a restricted form. We could require that all rules produce smaller continuations and evaluations, as was the case in $\mathcal{L}$. Such a requirement, though easy to automatically check, severely limits the variety and power of programming features that may be represented in S3OS. S4OS which is presented in the next chapter provides a more comprehensive solution to binding time analysis.

Finally, we observe that all of the transformations used to synthesize specifications do not disrupt the modularity of object language features described in S3OS. New language features may be added at any time, and the synthesized compiler and virtual machine can be extended. However, existing programs may need to be recompiled if the modification introduces new ways to trigger existing frames, or new ways to evaluate existing syntax.

Chapter 4

S4OS: Semicompositional S3OS

This chapter defines S4OS, a style of semantics specification that has stricter requirements than S3OS but is more versatile. It is used to substantiate the thesis statement using a style of specification which can be used to encode more expressive features than those permitted by S3OS. Section 4.1 motivates the transition from S3OS to S4OS by defining $\mathcal{L}_{\mathsf{fun}}$ which extends $\mathcal{L}$ with function application. Section 4.2 explains how the process of synthesizing the compiler and virtual machine can be ported from S3OS to S4OS. Section 4.3 concludes the chapter by demonstrating the flexibility of S4OS with a few features from C0.

4.1 Functions

In this section, we consider the implementation call-by-value function application for $\mathcal{L}_{\mathsf{fun}}$. Functions in C0 take lists of values as inputs. For simplicity, we only implement functions which accept a single argument. The grammar for $\mathcal{L}_{\mathsf{fun}}$ follows:

$\langle exp \rangle$ ::= $\langle numeric\text{-}constant \rangle$ | $\langle exp \rangle$ $\langle op \rangle$ $\langle exp \rangle$ | `(` $\langle fun\text{-}name \rangle$ $\langle exp \rangle$ `)`

$\langle op \rangle$ ::= `+` | `x` | ...

$\langle stm \rangle$ ::= `return` $\langle exp \rangle$ `;` | `let` $\langle var \rangle$ = $\langle exp \rangle$ `;` $\langle stm \rangle$

$\langle fun \rangle$ ::= $\langle fun\text{-}name \rangle$ `(` $\langle var \rangle$ `)` `{` $\langle stm \rangle$ `}`

$\langle prog \rangle$ ::= $\langle fun \rangle$*

Functions can be encoded as follows using HOAS:

```
vbl/f : (dest -> stm) -> valuable.
```

We also need a way supply function definitions to the specification. We create a new type of persistent resource that can be used to seed the CLF context:

```
src : valuable -> dest -> type. #mode src - -.
```

Functions are placed at specific destinations that we expect to be unique. These destinations serve as the function name that the grammar requires in each function definition. Function application can refer to the function to call using the destination. Listing 4.1 implements function application using these new definitions and previously discussed SSOS and S3OS techniques. With the

```
exp/app : dest -> exp -> exp.
f/app   : dest -> frame.

ev/app : eval (vbl/e (exp/app D E)) W
         -o { Exists x1. eval (vbl/e E) x1 *
              Exists x2. cont (f/app D) x1 x2 * cont f/dummy x2 W }.

tr/app : ret V X * cont (f/app D) X W * !src (vbl/f (\!x. S !x)) D
         -o { Exists x1. !ret V x1 *
              Exists x2. eval (vbl/s (S !x1)) x2 * cont f/dummy x2 W }.
```

Listing 4.1: Functions with HOAS

```
f : dest.

function : valuable =
  vbl/f (\!x. (stm/let (exp/op (exp/dest !x) op/+ e5) (\!y.
          (stm/let (exp/op (exp/dest !y) op/+ (exp/dest !y)) (\!z.
            (stm/return (exp/dest !z))))))).

function-decl : src function f.

#query * 1 * 1
Pi w1. Pi w2.
ret (val/e e10) w1 * cont (f/app f) w1 w2 -o { ret V w2 }.
```

Listing 4.2: Computing with source entry points

exception of matching on a function definition in the CLF context, the semantics for function application is not significantly different from that of let statements.

We generalize the encoding of function definitions with the following stylistic requirement:

S4OS requirement 4.1. *Any source term to which non-local control flow can occur must have a corresponding* source entry point *defined in the persistent context. These source entry points are called* `src` *in the CLF encoding. Each source entry point must be placed at a unique destination.*

Function calls require non-local control flow and their implementation must meet the given requirement. Listing 4.2 illustrates the use of source entry points to seed the CLF context and start evaluation or compilation. The program fragment from chapter 3 is now encoded as a function.

In order to continue compiling using the $\mathcal{L}_{\mathsf{fun}}$ spec, we need to know that continuations can still satisfy BSV. Since continuations are created using the program text in source entry points, source entry points must also be static. In the current specification, the only source entry points are those supplied when seeding the CLF context. The rules cannot produce new source entry points – the parser must directly produce them. Therefore, source entry points clearly satisfy BSV. Later, we will explain how to produce source entry points using forward inference rules and ensure BSV.

Continuations, are more complicated. Suppose the functions in the program text permitted recursion. `tr/app` produces a new continuation every time the body of a function is evaluated. Therefore, the number of continuations produced is dynamic. If the type system of the language permits general recursion (as is the case with C0), the number of continuations may not even be finite. However, we cannot avoid producing the continuation because of the S3OS requirement of distinct destinations for nonterminals. Without the requirement, Skolemization would substitute the same constructed destinations for multiple variables in let statements and function calls.

We address these problems by extending S3OS to a style called S4OS. The most significant difference is the elimination of HOAS and the introduction of *explicit contexts* that identify dynamic content. These explicit contexts are contexts of object language variables. We originally developed this style to model closures for an implementation of the λ calculus, so the listings refer to these contexts as closures (`clo`). We prefer the term *closure* to *explicit context* in this chapter to avoid confusion between the explicit context of object language bindings and the ubiquitous CLF contexts, but readers should note that these contexts are more general than closures in functional programming.

S4OS requirement 4.2. *HOAS is forbidden in CLF encodings of language features. Explicit closures and de Bruijn indices must be used instead to encode object language variables.*

Listing 4.3 introduces closures into the overarching SSOS infrastructure. Returned values depend on closures. The closure determines when a value is available for consumption and when a value can interact with other values. Evaluations must also refer to a closure so that they may consume or produce values in the correct closure. Continuations and source entry points are oblivious to closures.

S4OS requirement 4.3. *Evaluations and returned values depend on closures. Continuations are oblivious to closures.*

Listing 4.4 illustrates the implementation of functions without HOAS. The variable formerly introduced by a CLF lambda expression is now to be concretely provided by the parser. `ev/app` evaluates the argument to the function in the caller's closure. `tr/app1` creates a new closure in which the returned argument is copied, and the function body is evaluated. `tr/app2` copies the result of the function body back into the closure of the caller. Readers may find these rules reminiscent of the stack disciplined observed by procedural and sequential languages like C. We can use `clo-cons` to construct closures that have arbitrary graph structures. For simplicity, we present an example that resembles a stack.

S4OS requirement 4.4. *New closures may be related to existing closures using* `clo-cons`*. Every occurrence of* `clo-cons` *must be persistent to allow modular lookups.* [1]

Next, we need to update how we declare and access variables within these closures created by function application. Listing 4.5 also modifies let statements to eliminate HOAS. Multiple declarations of the same variable must now be prevented before a CLF encoding of the program

[1]Examples of features that need to look up closures in a modular way can be found in appendix C. For the purpose of C0, using linear `clo-cons` resources can be thought of as a global optimization that produces a familiar stack discipline.

```
clo   : type.
dest  : type.
eval  : valuable -> clo -> dest -> type.        #mode eval  - - -.
ret   : value -> clo -> dest -> type.           #mode ret   - - -.
cont  : frame -> dest -> dest -> type.          #mode cont  - - -.
cont1 : dest -> frame -> dest -> dest -> type. #mode cont1 - - - -.
src   : valuable -> dest -> type.               #mode src   - -.
```

Listing 4.3: Definition of closures

```
clo/cons : clo -> clo -> type. #mode clo/cons - -.

vbl/f   : dest -> stm -> valuable.
exp/app : dest -> exp -> exp.
f/app   : dest -> frame.
f/app2  : frame.

ev/app : eval (vbl/e (exp/app D E)) G W
         -o { Exists x1. eval (vbl/e E) G x1 *
              Exists x2. cont (f/app D) x1 x2 * cont f/dummy x2 W }.

tr/app1 : ret V G X * cont (f/app D) X W * !src (vbl/f X1 S) D
          -o { Exists g. !clo/cons g G * !ret V g X1 *
               Exists x2. eval (vbl/s S) g x2 * cont f/app2 x2 W }.

tr/app2 : ret V G' X * cont f/app2 X W * !clo/cons G' G
          -o { ret V G W }.
```

Listing 4.4: Functions with closures

is generated. An elaborator could rename variables that occur in the concrete syntax or reject programs that declare the same variable multiple times in the body of a function. C0 uses the latter strategy which we could similarly apply to $\mathcal{L}_{\mathsf{fun}}$. We could have used de Bruijn indices to encode variables, but to do so would be unnecessarily complex and inefficient for a first order language like C0. However, higher order examples using de Bruijn indices may be found in the appendix.

The rules to lookup variables ensure that we find the variable declarations associated with the current closure of evaluation. The assumption for $\mathcal{L}_{\mathsf{fun}}$ is that the closure and destination together can uniquely identify a returned value, even if features such a recursion allow a dynamic number of values to be associated with a destination. This assumption is only valid if we create a new closure every time we start evaluation at a source entry point. We make this assumption an S4OS requirement because it has applications in skolemization, which maps a potentially infinite number of existentially quantified variables to a finite set of constructed destinations.

S4OS requirement 4.5. *Every rule that matches on a source entry point must create a new closure in which any evaluations or values produced from the entry point can be placed.*

Listing 4.6 gives an example of CLF encoding without HOAS. Listing 4.7 illustrates another instance of closures constraining the returned values that may trigger a continuations.

```
stm/let : exp -> dest -> stm -> stm.
f/let   : dest -> valuable -> frame.

ev/let : eval (vbl/s (stm/let E D S)) G W
         -o { Exists x1. eval (vbl/e E) G x1 *
              Exists x2. cont (f/let D (vbl/s S)) x1 x2 *
              cont f/dummy x2 W }.

tr/let : ret V G X * cont (f/let D S) X W
         -o { !ret V G D * Exists x. eval S G x * cont f/dummy x W }.

exp/dest : dest -> exp.
f/dest   : dest -> frame.

ev/dest : eval (vbl/e (exp/dest D)) G W
          -o { Exists x. ret val/dummy G x * cont (f/dest D) x W }.

tr/dest : ret val/dummy G X * cont (f/dest D) X W * !ret V G D
          -o { ret V G W }.
```

Listing 4.5: `let` statements with closures

`tr/op2/add` requires that both values are in the same closure, but the continuation does not mention the closure.

We cannot yet prove that continuations and evaluations satisfy BSV, but we come very close to this idea because the S4OS specification evaluates source terms semicompositionally. Every rule produces continuations or evaluations using terms that are proper syntactic substructures of those which are matched in the antecedent, due to the lack of HOAS and substitution. The specification is semicompositional, but *not compositional* because the evaluations produced by rules may be proper syntactic substructures of the terms at source entry points, but not substructures of the terms in evaluations and continuations in the antecedent.

4.2 Synthesis for S4OS

The process of synthesizing a compiler and virtual machine from S4OS has the structure seen in figure 3.1. Every synthesized stage of S3OS specification has a corresponding stage in S4OS. First we eliminate existential quantifiers using skolemization by equality. Next, we use erasure to produce the compiler and VM. Code simplification is identical, so we do not present it again.

Skolemization

The underlying purpose of many of the design decisions of S4OS was to permit a concrete realization of destinations that satisfies BSV without losing precision. We achieve this goal by using the strategy from 3.2.1, but leaving in place all existential quantifiers that generate closures. The resulting style of specification is called S4OS-sko. Closures are handled later when transforming to S4OS-comp, because they are considered dynamic and subject to erasure.

```
f : dest. x : dest. y : dest. z : dest.

function : valuable =
  vbl/f x (stm/let y (exp/op (exp/dest x) op/+ e5)
           (stm/let z (exp/op (exp/dest y) op/+ (exp/dest y))
             (stm/return (exp/dest z)))).

function-decl : src function f.

#query * 1 * 1
Pi w1. Pi w2.
ret (val/e e10) w1 * cont (f/app f) w1 w2 -o { ret V w2 }.
```

Listing 4.6: Listing 4.2 without HOAS

```
tr/op1 : ret V1 G X * cont (f/op1 B E2) X W
         -o { Exists x1. ret V1 G x1 * Exists x2. eval E2 G x2 *
              cont1 x1 (f/op2 B) x2 W }.

tr/op2/add : ret (val/e (exp/num N1)) G X1 * ret (val/e (exp/num N2)) G X2 *
             cont1 X1 (f/op2 op/+) X2 W * !bin/add N1 N2 b0 N3 _
             -o { ret (val/e (exp/num N3)) G W }.
```

Listing 4.7: Arithmetic operations with closures

Listing 4.8 illustrates skolemization by equality on the rules for function application. It also handles evaluations arising from source entry points in an interesting way – they are always placed at the same destination constructed from the destination of the source entry point. We require this property for compilation.

The proof of equivalence of the S4OS-sko specification is similar to the bisimulation proof given for theorem 3.1. The theorem is extended below to account for closures.

Definition 4.1. *Suppose that:*

- Ω *is a set of destinations generated by quantifiers.*
- Ξ *is a set of closures generated by quantifiers.*
- Γ *is a state containing linear and persistent resources that may mention destinations in* Ω*.*

A S4OS-sko state (Δ, Ξ') *is said to be a* realization *of* (Γ, Ω, Ξ) *if and only if there exists* σ *and* κ *such that* $\Delta = \sigma(\kappa\Gamma)$ *where*

- σ *is a substitution of destinations constructed using constructors from the S4OS-sko signature for variables in* Ω*.*
- κ *is a substitution of variables from* Ξ' *for variables from* Ξ*, based on a one-to-one mapping between* Ξ *and* Ξ'*.*

Theorem 4.1. *A S4OS specification can produce a trace* Θ *when running an input program if and only if the corresponding S4OS-sko specification can produce a trace* Θ' *such that for all* n*, the* n^{th} *state in* Θ' *is a realization of the* n^{th} *state in* Θ*.*

A sketch of the proof can be found in appendix A.2.1.

```
d/10    : dest -> dest.
d/11    : dest -> dest.
d/12    : dest -> dest.

ev/app : eval (vbl/e (exp/app D E)) G W
         -o { eval (vbl/e E) G (d/10 W) *
              cont (f/app D) (d/10 W) (d/11 W) * cont f/dummy (d/11 W) W }.

tr/app1 : ret V G X * cont (f/app D) X W * !src (vbl/f X1 S) D
          -o { Exists g. !clo/cons g G * !ret V g X1 *
               eval (vbl/s S) g (d/12 D) * cont f/app2 (d/12 D) W }.

tr/app2 : ret V G' X * cont f/app2 X W * !clo/cons G' G
          -o { ret V G W }.
```

Listing 4.8: Skolemized function semantics

```
eval  : valuable -> dest -> type.               #mode eval  - -.
ret   : dest -> type.                           #mode ret   -.

ev/app : !eval (vbl/e (exp/app D E)) W
         -o { !eval (vbl/e E) (d/10 W) *
              !cont (f/app D) (d/10 W) (d/11 W) *
              !cont f/dummy (d/11 W) W }.

tr/app1 : !ret X * !cont (f/app D) X W * !src (vbl/f X1 S) D
          -o { !ret X1 * !eval (vbl/s S) (d/12 D) *
               !cont f/app2 (d/12 D) W }.

tr/app2 : !ret X * !cont f/app2 X W -o { !ret W }.
```

Listing 4.9: Compilation of functions

Compiler

We synthesize a S4OS-comp specification from a S4OS-sko specification by making all continuations, evaluations and returned values persistent, and then erasing all dynamic terms. Values *and closures* are removed from `ret`. Closures are also removed from `eval`. Closure links (`clo-cons`) are also removed. Any term that becomes ill-moded is removed by mode-driven erasure.

An additional step is required to synthesize a compiler from S4OS – any remaining existential quantifier which generates an unused variable is also erased. This step handles the quantifiers which generate closures which were not eliminated by skolemization. Listing 4.9 illustrates the S4OS-comp version of rules specifying procedures.

The following theorems prove that the resulting compiler can terminate by saturation:

Theorem 4.2. *S4OS-comp specifications are guaranteed to saturate.*

We do not give a complete proof, but give a sketch of the reasoning – S4OS-sko generates one

```
eval   : clo -> dest -> type. #mode eval - -.
src    : dest -> type.        #mode src  -.

ev/app : eval G W * cont (f/app D) (d/10 W) (d/11 W) *
         cont f/dummy (d/11 W) W
         -o { eval G (d/10 W) }.

tr/app1 : ret V G X * cont (f/app D) X W * !src D
          -o { Exists g. !clo/cons g G * !ret V g X1 *
               eval g (d/12 D) * cont f/app2 (d/12 D) W }.

tr/app2 : ret V G' X * cont f/app2 X W * !clo/cons G' G
          -o { ret V G W }.
```

Listing 4.10: Virtual instructions for functions

finite set of destinations for all resources constructed from subterms of a source entry point. Since there are finitely many source entry points, the destinations satisfy BSV. BSV of destinations, semicompositionality of static content, and erasure of dynamic content is sufficient to prove saturation.

Theorem 4.3. *A S4OS-comp specification is guaranteed to collect a set of continuations that is a superset of that which could have been collected by S4OS-sko.*

Again, we omit the complete proof. However, we observe that erasing the dynamic terms only increases the number of possible firings of rules, so the general outline of the proof of theorem 3.3 can be reused.

Virtual Machine

The techniques for synthesizing a virtual machine are also similar to those used for S3OS. Continuations are made persistent to synthesize S4OS-JIT from S4OS-sko. Next, evaluation rules are inverted so that continuations occur in the antecedent instead of the consequent. Trigger rules no longer produce continuations. Instead, the continuations from the compiler are used to seed the context. Valuable terms are removed from evaluations *and source entry points*. The resulting style of specification is called S4OS-VM. S4OS-VM can further be run through code simplification.

Listing 4.10 explains how source entry points are handled, using function application as an example. In a language like C0, it is possible to completely eliminate `src` from the virtual machine. In languages whose type discipline permits calls to functions which may not have a definition, we need to know that a valid source entry point exists at the matched destination, even if the program text has been erased from it. For instance, methods in smalltalk and other features involving dynamic dispatch require source entry points without the program text to be preserved.

The definitions, and statements of validity of S4OS-JIT and S4OS-VM have been ported from chapter 3 below:

Definition 4.2. *A S4OS-JIT state Δ, Ξ' is called a* compiling instantiation *of a S4OS-sko Γ, Ξ if*

and only if:

- Δ *contains the same multiset of evaluations as* $\kappa\Gamma$
- Δ *contains the same multiset of returns as* $\kappa\Gamma$
- Δ *contains a superset of the continuations in* $\kappa\Gamma$ *made persistent.*

κ *is a substitution of variables from* Ξ' *for variables from* Ξ*, based on a one-to-one mapping between* Ξ *and* Ξ'*.*

Theorem 4.4. *A S4OS-sko specification can produce a trace* Θ *when evaluating a program if and only if the corresponding S4OS-JIT specification can produce a trace* Θ' *such that for all* n*, the* n^{th} *state of* Θ' *is a compiling instantiation of the* n^{th} *state of* Θ*.*

Definition 4.3. *A S4OS-VM state* Δ *is called an* ahead-of-time instantiation *of a S4OS-JIT state* Γ *if and only if:*

- Δ *contains the same multiset of evaluations as* $\kappa\Gamma$*, but with valuable terms erased.*
- Δ *contains the same multiset of returns as* $\kappa\Gamma$
- Δ *contains all the continuations that could possibly be generated by an S4OS-JIT specification when run on* $\kappa\Gamma$*, and making all the continuations persistent.*

κ *is a substitution of variables from* Ξ' *for variables from* Ξ*, based on a one-to-one mapping between* Ξ *and* Ξ'*.*

Theorem 4.5. *A S4OS-JIT specification can produce a trace* Θ *when evaluating a program if and only if the corresponding S4OS-VM specification can produce a trace* Θ' *such that for all* n*, the* n^{th} *state of* Θ' *is an ahead-of-time instantiation of the* n^{th} *state of* Θ*.*

We omit proofs for these properties because they port the structure of the corresponding S3OS properties, as did theorem 4.1.

4.3 Semicompositionality and Loops

This section defines $\mathcal{L}_{\mathsf{oop}}$, which extends $\mathcal{L}_{\mathsf{fun}}$ with mutable automatic variables and loops. These features illustrate how another form of repetition in S4OS. Mutable variables are added into the mix because they allow us to express interesting computations with loops. $\mathcal{L}_{\mathsf{oop}}$ modifies the $\mathcal{L}_{\mathsf{fun}}$grammar as follows:

```
⟨stm⟩  ::= ...
        |  set ⟨var⟩ = ⟨exp⟩ ; ⟨stm⟩
        |  while ( ⟨exp⟩ ) { ⟨stm⟩ } ; ⟨stm⟩
        |  nop
```

A few S4OS requirements and theorem statements are modified slightly to accommodate the new features, but the changes do not detrimentally affect the ability to synthesize a compiler and VM.

```
stm/set : exp -> dest -> stm -> stm.
f/set   : dest -> valuable -> frame.

ev/set : eval (vbl/s (stm/set E D S)) G W
         -o { Exists x1. eval (vbl/e E) G x1 *
              Exists x2. cont (f/set D (vbl/s S)) x1 x2 *
              cont f/dummy x2 W }.

tr/set : ret V G X * cont (f/set D S) X W * ret _ G D
         -o { ret V G D * Exists x. eval S G x * cont f/dummy x W }.
```

Listing 4.11: `set` statements

```
#query * 1 * 1
Pi w1. Pi w2.
ret (val/e e10) w1 * cont (f/app f) w1 w2 -o { ret V w2 }.
```

Listing 4.12: Query that fails due to unconsumed resources

```
tr/let : ret V G X * cont (f/let D S) X W
         -o { @ret V G D * Exists x. eval S G x * cont f/dummy x W }.

tr/app1 : ret V G X * cont (f/app D) X W * !src (vbl/f X1 S) D
          -o { Exists g. !clo/cons g G * @ret V g X1 *
               Exists x2. eval (vbl/s S) g x2 * cont f/app2 x2 W }.

tr/set : ret V G X * cont (f/set D S) X W * @ret _ G D
         -o { @ret V G D * Exists x. eval S G x * cont f/dummy x W }.

tr/dest : ret val/dummy G X * cont (f/dest D) X W * @ret V G D
          -o { @ret V G D * ret V G W }.
```

Listing 4.13: Garbage collected mutable state

4.3.1 Mutable Variables and Automatic Storage

$\mathcal{L}_{\mathsf{fun}}$ variables are defined once and their values remain in the persistent context indefinitely. Suppose we wish to implement mutable local variables in the style of C0. The first step in the specification of $\mathcal{L}_{\mathsf{oop}}$ would be to make the value associated with the variable linear so that it can be consumed during mutation and be replaced by a new variable associated with the destination. Listing 4.11 implements this change to local variables.

Removing persistence may be sufficient to compute the required results, but it does not specify when to release the memory dedicated to variables. This shortcoming can be observed when running a program using a Celf query such as the one in `gc-query`. Linear resources must be consumed, otherwise they must be listed in the consequent of the query along with the expected returned value. The query fails because the specification does neither.

One solution is to emulate some form of garbage collection using the *affine* modality, which is written as `@` in CLF. Affine resource can be used at most once. Linear resources must be used exactly once. Persistent resources can be used zero or more times. Therefore, the affine modality

```
f/del   : dest -> frame.

ev/let : eval (vbl/s (stm/let E D S)) G W
         -o { Exists x1. eval (vbl/e E) G x1 *
              Exists x2. cont (f/let D (vbl/s S)) x1 x2 *
              cont (f/del D) x2 W }.

tr/let : ret V G X * cont (f/let D S) X W
         -o { ret V G D * Exists x. eval S G x * cont f/dummy x W }.

tr/del : ret V G X * cont (f/del D) X W * ret _ G D
         -o { ret V G W }.
```

Listing 4.14: Automatic variables on the stack

```
ev/app : eval (vbl/e (exp/app D E)) G W
         -o { Exists x1. eval (vbl/e E) G x1 *
              Exists x2. cont (f/app D) x1 x2 * cont f/dummy x2 W }.

tr/app1 : ret V G X * cont (f/app D) X W * !src (vbl/f X1 S) D
          -o { Exists g. clo/cons g G * ret V g X1 *
               Exists x2. eval (vbl/s S) g x2 *
               Exists x3. cont (f/del D) x2 x3 * cont f/app2 x3 W }.
```

Listing 4.15: Functions allocate automatic variables

is ideal for encoding mutable garbage collected state. Listing 4.13 modifies all rules that access variables so that they all mark variables as garbage collected.

Observation 4.1. *It is possible to imitate garbage collection in CLF specifications. Any immutable resource that must be garbage collected can be made persistent. Any mutable resource to be garbage collected can be made affine.*

Another solution is to implement the C0 stack discipline. The lifetime of a variable in $\mathcal{L}_{\text{oop}}$ and C0 is tied to its lexical scope, beyond which a value cannot escape. Listing 4.14 cleans up variables at the end of their scope by introduce a continuation at the end of the list of statements with the frame `f/del`. Function arguments are also managed in the same way in listing 4.15.

Mutable state fits in well with the synthesis techniques that have already been discussed. However, during compilation, affine resources must also be converted to persistent resources for saturation.

4.3.2 While Loops

The `while` loop from C0 is another example of a feature which involves repeated execution of code generated from one part of the program text. We could not define loops in S3OS because our techniques for VM synthesis for S3OS could not handle features that involved repetition. We can now implement `while` loops in S4OS. Loops are also interesting to examine because they do not require non-local control flow to execute, so we can state their semantics without creating

```
stm/while : exp -> stm -> stm -> stm.
f/while1  : valuable -> valuable -> valuable -> frame.
f/while2  : valuable -> valuable -> valuable -> frame.

bin/zero : bin -> bit -> type.
#mode bin/zero + -.

ev/while : eval (vbl/s (stm/while E S1 S2)) G W
           -o { Exists x. eval (vbl/e E) G x *
                cont (f/while1 (vbl/e E) (vbl/s S1) (vbl/s S2)) x W }.

tr/while1/z : ret (val/e (exp/num N)) G X * cont (f/while1 E S1 S2) X W *
              bin/zero N b0
              -o { Exists x. eval S2 G x * cont f/dummy x W }.

tr/while1/s : ret (val/e (exp/num N)) G X * cont (f/while1 E S1 S2) X W *
              bin/zero N b1
              -o { Exists x. eval S1 G x * cont (f/while2 E S1 S2) x W }.

tr/while2/nop : ret (val/s stm/nop) G X * cont (f/while2 E S1 S2) X W
                -o { Exists x. eval E G x * cont (f/while1 E S1 S2) x W }.

tr/while2/ret : ret (val/s (stm/return V)) G X * cont (f/while2 E S1 S2) X W
                -o { ret (val/s (stm/return V)) G W }.
```

Listing 4.16: While loop semantics

source entry points.

Listing 4.16 defines a simple form of loop. For simplicity, it does not support control features such as `break` or `continue` that are commonly found in imperative languages, but these features can be added easily. The CLF term `(stm/while E S1 S2)` is to be read as:

```
while (E) {S1}; S2;
```

in C0. The $\mathcal{L}_{oop}$ fragment does not support boolean values, which are used in C0. Instead, the loop guard evaluates to an integer as one would expect in C, and the loop body is executed if the guard evaluates to a nonzero value.

In order to support loops, we had to change the semantics of statements too, because a list of statements (such as those in a loop body) does not have to return a result. A list of statements can be terminated by a `stm/nop`. Listing 4.17 describes the modified semantics for statements. We also introduce a new invariant – evaluating a statement produces a *statement value* for which there is the corresponding `val/s` constructor. No-ops and return statements are considered the two canonical values for statements. The other statements retain their CPS-like behavior.

The semantics for loops relies on two nonterminal frames – `f/while1` waits for the result of the loop guard, and `while2` waits for the result of the loop body. These nonterminals challenge a S4OS requirement that nonterminals must be placed at fresh destinations. In fact, the trigger rules cannot create fresh destinations because the number of destinations and continuations created would depend on the number of iterations of the loop, violating BSV.

We continue using the techniques that have already been developed for synthesis, but account

```
val/s       : stm -> value.
stm/return : exp -> stm.
f/return    : frame.

ev/return : eval (vbl/s (stm/return E)) G W
            -o { Exists x. eval (vbl/e E) G x * cont f/return x W }.

tr/return : ret (val/e V) G X * cont f/return X W
            -o { ret (val/s (stm/return V)) G W }.

stm/nop : stm.
f/nop    : frame.

ev/nop : eval (vbl/s stm/nop) G W
         -o { Exists x. ret val/dummy G x * cont f/nop x W }.

tr/nop : ret val/dummy G X * cont f/nop X W
         -o { ret (val/s stm/nop) G W }.
```

Listing 4.17: Modified statement semantics

```
tr/app2 : ret (val/s (stm/return V)) G' X * cont f/app2 X W * clo/cons G' G
          -o { ret (val/e V) G W }.
```

Listing 4.18: Modified function return

for the new behavior of nonterminals by appealing to the notion of the *size* of a frame.

Definition 4.4. *Any two frames* `F1` *and* `F2` *are of* equal size *if there exists a configuration of dynamic resources such that both of the following are true:*

- *Given a continuation with* `F1`*, there exists a sequence of forward inference rules which can be applied to produce a continuation with* `F2` *and positionally equivalent terms.*
- *Given a continuation with* `F2`*, there exists a sequence of forward inference rules which can be applied to produce a continuation with* `F1` *and positionally equivalent terms.*

`F2` *is considered smaller than* `F1` *if there only exists a sequence of forward inference rules from* `F1` *to* `F2`*, and the terms in* `F2` *are not bigger than those in* `F1`*.*

We can also define continuations to have the same size if and only if the frames they contain have the same size.

S4OS requirement 4.6. *Any trigger rule which produces a continuation of the same size as the continuation matched in the antecedent must ensure that both continuations are placed at the same target destination* `W`*. If the continuations are nonterminals of different size, a fresh destination must be created.*

Clearly, the frames used to define loops are the only frames in our language specification which are of equal size, and they meet the given requirement. For the most part we do not have to revise any of the definitions or proofs given so far to accommodate this behavior. Lemma A.2.1 which is used in many of the proofs has to be slightly altered slightly, as stated in Lemma

A.2.3 in the appendix.

The intuitive reason for the correctness of the size-based reasoning is the skolemization strategy used in S4OS-sko, which maintains the following properties:

- Continuations of the same size using different frame constructors are guaranteed to wait on different destinations, because we use distinct destination constructors. Therefore, there is no ambiguity in identifying which of many continuations of the same size may be activated.
- Continuations of the same size using the same frame constructor are indistinguishable in their effects, so it is not a problem for them to wait on the same destination.

Saturation of S4OS-comp is also not affected. There are only finitely many frame constructors of the same size and all continuations of the same size are placed at the same destination. Therefore, there are only finitely many continuations of the same size.

The modified S4OS requirements in this section aren't intended as a special case to handle while loops. They permit us to encode other features such as efficient sequential initialization of arrays, which we use in our implementation of C0.

Chapter 5

Conclusion

Section 5.1 reviews the thesis statement and our contributions towards substantiating it. Section 5.2 gives a high-level overview of our implementation of C0 [Pfe11] in S4OS. Section 5.3 considers future directions of research. Section 5.4 surveys related work in compilation for other styles of semantic specification.

5.1 Contributions

Let us briefly review the thesis statements and our contributions towards them:

Thesis Statement 1. *The substructural operational semantics of a programming language can be stated in such a way that the latent resources are of static provenance, and passive resources are dynamic values. In such a form, the latent resources correspond to instructions synthesized from the program, and passive resources correspond to operands to these instructions.*

Towards the statement above, we presented S3OS and S4OS which separated static and dynamic content into continuations and returns respectively. The continuations are the latent resources that correspond to instructions.

Thesis Statement 2. *Given a separable SSOS specification for a programming language, the latent resources/instructions for a program can be computed without referring to the program's inputs or dynamic values. We can do so by logically approximating the behavior of the specification.*

In support of the statement above, we present linear logical approximations such as persistence, skolemization and mode-driven erasure that can collect all the continuations without any reference to dynamic inputs. S4OS-sko and S4OS-comp are the synthesized specifications which apply these techniques.

Thesis Statement 3. *A separable SSOS specification can be used to synthesize a Virtual Machine that executes the virtual instruction set characteristic of the object language. The virtual machine serves as a specification of the instruction set, and it does not refer to the original program text.*

In support of the statement above, we present S4OS-JIT, S4OS-VM, and the code simplifier

which can execute the continuations generated by S4OS-comp after erasing program text.

Utility of the Virtual Instruction Set

As we have previously explained, our techniques do not compile an object language to any arbitrary target language. Instead, S4OS-VM and S4OS-comp manipulate a virtual instruction set synthesized from the language specification. Some natural questions arise: How much does this instruction set simplify the process of compiling for another desired target instruction set? Is it a suitable intermediate language for optimizing and then targeting other instruction sets? How much work remains for a compiler implementer to translate from the virtual instruction set to another target machine?

Typically, a language developer choosing to implement a compiler does so to gain the performance benefits of a state machine for which efficient hardware implementations exist. This state machine typically has instructions which cause state transitions, and some of these transitions are considered atomic. When translating a source language to a target instruction set, the compiler implementer must determine exactly how to break down an input program into atomic operations and eliminate program text. Our techniques largely focus on helping language designers with this stage of development. SSOS obliges developers to specify the granularity of object language features in the semantic specification by forcing a choice between a forward chaining and backward chaining rules. Our techniques use this information to synthesize an instruction set that captures this granularity of operations while separating out dynamic content and eliminating program text.

The instruction set is accompanied by a precise definition of the behavior of each instruction. Since the definition take the form of forward inference rules, they can also be run as a virtual machine in Celf, thereby simplifying testing. For developers of languages with concurrency, parallelism or nondeterminism, the virtual instruction set precisely details when synchronization is required.

We have not tested the suitability of S4OS-VM as an intermediate representation for an optimizing compiler framework. However, we can hypothesize about its potential uses by comparing it to two compiler frameworks – LLVM [Lat02] and ML-RISC [GL00].

LLVM (Low Level Virtual Machine) is an optimizing compilation infrastructure that operates on programs implemented in a virtual instruction set called the LLVM IR. LLVM IR is largely designed as a one-stop intermediate representation. A language designer simply has to convert an object language to the LLVM IR, and the LLVM backend takes care of optimizing and targeting any supported hardware platform. The LLVM IR is not target independent however, and language developers wishing to target a hardware platform not supported by LLVM cannot use it.

S4OS-VM is not an intermediate representation in this sense of LLVM because there is no single S4OS-VM instruction set that multiple languages can target. Rather, it is a family of instruction sets with a few overarching characteristics such as intrinsic support for concurrent computation and mutation. The specific instruction set depends on the definition provided by the language designer. If multiple languages have overlapping features, their S4OS-VM instructions sets may also have significant overlap.

This structure of S4OS-VM is more reminiscent of ML-RISC, which also does not provide a single virtual instruction set. Instead, ML-RISC operates on an abstraction called a *flow graph* to

perform truly target independent transformations. The compiler developer is also responsible for providing a translation from flow graphs to hardware targets. Therefore, S4OS-VM may have applications similar to those of ML-RISC.

Testing

We tested the practicality of S4OS by specifying C0. We have illustrated many of our results by stripping down C0 features to derive $\mathcal{L}$, $\mathcal{L}_{\mathsf{fun}}$ and $\mathcal{L}_{\mathsf{oop}}$.

In order to fully illustrate encodings of higher order features using explicit closures in S4OS, we also implemented call-by-name lambda calculus and futures. We have included the implementation of the latter in appendix C for advanced readers. The implementation is more complex than the specification presented in [PS09] for similar features. Nonetheless, it demonstrates that S4OS and our synthesis techniques can scale up to call-by-name and concurrent features.

5.2 S40S for C0

We give only a high level overview of our implementation of the C0 specification. We leave it to future work to fully describe the implementation details. The software implemented as a part of the project includes:

- `cc0-CLF`: A modified version of the reference C0 compiler which parses and type-checks input code but outputs a CLF encoding of the program instead.
- `lolcpu` (LOgicaL CPU): A reference implementation of the 32-bit ALU whose existence is assumed by the C0 specification. This specification consists entirely of backward chaining rules.
- The S4OS for the CLF encoding of C0: Interesting C0 features that were implemented and tested but not discussed in this dissertation are dynamic allocations, pointers, arrays, structures and exceptions. We excluded the following:
    - The contract and annotation language: Contracts in C0 are boolean expressions which have access to a few specially defined variables and cause the program to abort immediately if they evaluate to false.
    - Characters and strings: Their implementations are similar to those of fixed width integers and arrays.
    - Standard library functions.
- Hand generated S4OS-sko, S4OS-comp, S4OS-JIT and S4OS-VM versions of the C0 specification.
- A test harness to run the various S4OS specifications of C0 on the regression test suite maintained with the reference compiler.

As of this writing, Celf is in active development and does not support saturation. As a result, the S4OS-comp specification is completely untested. Instead, to test the other stages, we copied the continuations generated by the S4OS-JIT specification and seeded the S4OS-VM context with them.

The virtual instruction set generated by S4OS-JIT is very similar to the instruction in the reference virtual machine (C0VM) [Pfe12a]. However, we could not find any construct in C0VM that was analogous to the skolemized destinations. C0VM was designed only with the intention of representing completely sequential operations on a stack machine. The assumptions available to C0VM are in general not available to S4OS-VM, which must accommodate the modular addition of concurrent and nondeterministic features. As a result, we could not directly map the instructions generated by S4OS-comp to their counterparts in C0VM that pushed and popped arguments from a stack. However, the result of performing register allocation on the skolemized destinations was similar to output produced by cc0, the end-to-end compiler.

5.3 Future Work

The following are possible directions for future work that have not already been mentioned.

Mechanize the Synthesis of Specifications

The main challenge with complete mechanization of the synthesis of S4OS-sko, S4OS-comp, S4OS-JIT and S4OS-VM is the lack of a formal definition of what it means for a specification to conform to S4OS, or even SSOS. This dissertation continues the tradition established by prior literature explaining SSOS as a set of techniques [Pfe04][Pfe06][PS09]. There is no single formal definition of SSOS, and consequently our definition of S4OS is also quite flexible.

Notwithstanding the lack of a strict definition, we have endeavored to provide guidance towards mechanization. The proof sketches in appendix A serve as a reasonably detailed record of the assumptions that need to be true should we attempt a precise definition. Given such a definition, an interesting future direction for research would be to mechanize the bisimulation proofs from the appendix in a logical framework such as SLS [Sim12] or CLF [WCPW03, CPWW03]. To the best of our knowledge, bisimulation proofs have not yet been encoded in logical frameworks. However, it is the subject of active research[Cer].

Implement Memory Management

In this dissertation, we have presented S4OS and S4OS-VM without much consideration for efficient memory management, which is an important aspect of compiler generation. At the very least, synthesized virtual machines must support garbage collection if the object language needs it. Ideally, the virtual instruction set would be able to provide information such as garbage collection roots and valid preemption points that can help realize a garbage collector in any environment that we target from S4OS-VM.

To the best of our knowledge, existing literature in compiler generation for other styles of semantic specification does not directly address the problem of memory management and garbage collection. Partial evaluation can partly solve the problem when the specification language for the interpreter is garbage collected, because residual programs are specification language computations that stand for object language computations, and therefore have access to the garbage collected store of the specification language. This approach usually requires more input from the

designer of the interpreter to separate manually managed state and garbage collected state, thus detracting from the simplicity offered by semantic specifications.

We believe there is potential for interesting future work in garbage collection for S4OS specifications, because the persistent and affine modalities allow us to encode memory management as a first class citizen of the semantics specification.

5.4 Related Work

To conclude the dissertation, we cite some of the prior work in semantics-driven compiler generation that we did not cite in the previous chapters but have subtly influenced the development of this dissertation. The works cited here are not intended to be an exhaustive list of publications in their respective areas.

Partial Evaluation for Denotational Semantics

Denotational interpreters for simple procedural languages have been transliterated successfully into functional languages and partially evaluated to produces a compiler that reasonably eliminates the overhead of interpretation. Jorgensen has presented work in untyped programming languages such as Scheme [Jør91]. Danvy has presented work applicable to typed environments such as SML [DV96].

Compiler Generation for Action Semantics

[Lee87, Ørb94] are examples of work demonstrating compiler generation for Action Semantics [Mos96]. These works have focused more on implementing optimizations in synthesized compilers to improve the quality of code.

Compilation through Semantic Correspondence

A compiler may be generated automatically or semi automatically by translating the semantic interpreter. Some work such as [JS80] used a denotational semantics to target a universal program representation which was easy to optimize and translate to machine code.

There was also some research on techniques that eschewed universal representations, and instead sought a virtual instruction set that preserved the structure of the input semantic specification. For instance, Ager et al. explained how techniques such as CPS conversion and defunctionalization could be applied to interpreters written in functional programming languages to synthesize a virtual instruction set and a compiler targeting it [ABDM03b, ABDM03a]. Our research was significantly inspired by these techniques.

Appendix A

Proofs Sketches of Theorems

The informal proofs in this chapter rely on contradiction to state that there is no rule which violates a particular pattern. If we were to mechanize these proofs formally, we could not rely on informal descriptions of patterns which the rules satisfy. Therefore, the appeals to contradiction would be replaced by case analysis on each rule in the language specification to ensure that it satisfies the require property. The properties required of rules would likely take the form of judgments. Such a formalization is beyond the scope of this dissertation, but we hope that the following informal outlines provide useful direction towards formalization in future work.

A.1 Properties from Chapter 3

A.1.1 Proof of Theorem 3.1 from Section 3.2.1

Lemma A.1.1. *Any two nonterminals are placed at different destinations by a skolemized S3OS specification, unless they are constructed from positionally equivalent terms in the program text, using the same frame constructors.*

Proof. This lemma can be proved as a state invariant. The initial state satisfies the invariant because it contains only one nonterminal. We can handle states with multiple nonterminals by generating distinct atomic propositions using quantifiers. Given a state that satisfies the invariant, it remains to be established that every application of a forward inference rule maintains the invariant.

We know that a new nonterminal `N2` can only be produced when a rule `R` matches some existing nonterminal `N1` at target destination `W`. The skolemization strategy places `N2` at a destination of the form `(d/x W)`.

Assume for the sake of contradiction that there exists another nonterminal `N4` at `d/x W` that was constructed from source terms that are positionally nonequivalent to those in `N2`. `N4` would have to be placed there by some rule matching `N3` at `W` and also constructing the new destination using `d/x`. Since we use distinct destination constructors in each rule and do not destruct destinations in any rule, there exists only one such rule – `R`. Two cases are possible:

- `N1` and `N3` are constructed from terms that have different occurrences in the program text. In this case `N1` and `N3` being placed at `W` contradicts the state invariant before the production of `N2`.
- `N1` and `N3` are constructed from terms that are positionally equivalent. In this case, the rule `R` constructed `N2` and `N4` from subterms of positionally equivalent terms or substitution of the same destination into positionally equivalent lambda expressions. This contradicts the assumption that `N2` and `N4` were constructed from positionally nonequivalent terms.

□

Lemma A.1.2. *Suppose we have an S3OS specification that substituted destinations generated by an existential quantifier into source expressions. After skolemization, any two positionally nonequivalent lambda binders in the source are guaranteed to have different constructed destinations substituted for the bound variables.*

Proof. Any rule that substitutes a destination into a term has the following structure:

```
[0 or more triggering resources] * <nonterminal (\!d:dest.  E !d) W>
-o { resource mentioning (E !(d/x W)) }.
```

positionally nonequivalent lambda terms would have to originate in nonterminals containing positionally nonequivalent source terms. We already know from lemma A.1.1 that such nonterminals are placed at different destinations. Therefore, the derived destinations `d/x W` are also guaranteed to be different. □

Lemma A.1.3. *Any two continuations produced by a skolemized S3OS specification must wait on different destinations, unless the continuations contain frames constructed from positionally equivalent source terms using the same constructor.*

Proof. Consider an arbitrary pair of continuations with positionally different source terms. The following cases are possible:

- Both continuations were used to seed the state before running the specification. This does not occur in any scenario we have discussed so far, but we can handle it by defining distinct atomic destinations for the continuations to wait on.
- One of the continuations is from the initial state, and the other was generated by a rule. In this case, they are guaranteed to wait on different destinations because the continuation from the initial state waits on an atomic destination whereas the generated continuation waits on a destination of the form `d/x W`.
- Both continuations are generated by rules. There are three sub-cases to consider:
  - The continuations are generated by different rules. Since each rule uses different destination constructors for skolemization, the continuations cannot possibly wait on the same destination.
  - The continuations are different resources generated by the same rule, i.e. there exists a rule `nonterminal -o { K1 * K2 * ...}` and the two continuations are respectively instances of `K1` and `K2`. S3OS requires that the two continuations wait on destinations that are bound separately, and our skolemization process uses distinct destination constructors for each quantified variable. Therefore, the continuations cannot wait on the same destination, as above.

▪ The two continuations are instances of the same resource generated by a rule, i.e. there exists a rule `nonterminal -o { K * ... }`, and the continuations are both instances of `K` that were generated by two different firings of the rule (but they are not positionally equivalent). The continuations respectively wait on constructed destinations of the form `d/x W1` and `d/x W2`, where `W1` and `W2` are respectively the target destination of the antecedent in each firing of the rule. The two continuations wait on different destinations if and only if `W1` and `W2` are different. This requirement is met because two positionally nonequivalent nonterminals cannot have the same target destination (inverse of lemma A.1.1).

□

The above lemma is easily generalized to `cont1` because we use different constructors for each destination being waited on. The following completes the proof of bisimulation between S3OS and S3OS-sko (theorem 3.1):

Proof. The initial state of both traces are equivalent because we seed the state using the same query. If two states are equivalent, then one is trivially a generalization of the other using zero substitutions. We wish to prove the following trace invariant:

Suppose after n steps that the S3OS computation is currently in a state Γ, and the S3OS-sko computation is currently in a state Δ such that Δ is a realization of Γ.

- For any rule in the S3OS specification that causes a state transition $\Gamma \to \Gamma'$, there exists a corresponding rule in S3OS-sko causing a transition $\Delta \to \Delta'$ such that Δ' is a realization of Γ'. This is the *soundness* direction.
- For any rule in the S3OS-sko specification that causes a state transition $\Delta \to \Delta'$, there exists a corresponding rule in S3OS causing a transition $\Gamma \to \Gamma'$ such that Δ' is a realization of Γ'. This is the *completeness* direction.

First, the soundness direction. Let $\Delta = \sigma\Gamma$. Suppose a rule R_1 from the S3OS specification matches some antecedent $A \subset \Gamma$. We know that R_1 will also match σA because R_1 only uses unification variables to refer to destinations in the antecedent. Any terms that were equivalent before substitution should remain equivalent after substitution. Since the skolemized version $R2$ matches the same antecedent, it can fire on $\sigma A \subset \sigma\Gamma$. Therefore R_2 form the S3OS-sko specification can match on Δ.

Suppose R_1 produces consequent C, i.e. $\Gamma' \equiv (\Gamma \backslash A, C)$. Then R_2 produces some $\sigma'\sigma C$, where σ' is simply the set of constructed destinations substituted when skolemizing existentially quantified variables. We can say the following so far:

$$\Delta' \equiv (\Delta \backslash \sigma A, \sigma'\sigma\ C) \equiv (\sigma\Gamma \backslash \sigma A, \sigma'\sigma C) \equiv (\sigma(\Gamma \backslash A), \sigma'\sigma C)$$

Since σ' substitutes for variables which did not exist in $\sigma\Gamma$, it is safe to apply it:

$$(\sigma(\Gamma \backslash A), \sigma'\sigma C) \equiv (\sigma'\sigma(\Gamma \backslash A), \sigma'\sigma C) \equiv \sigma'\sigma(\Gamma \backslash A, C) \equiv \sigma'\sigma(\Gamma')$$

This means that $\Delta' \equiv \sigma'\sigma\Gamma'$, and Δ' is a realization of Γ'.

Next, the completeness direction. Let $\Delta = \sigma\Gamma$. Suppose a rule R_2 from the S3OS-sko specification matches some antecedent $\sigma A \subset \sigma\Gamma$. We need to know that the version of R_1 without Skolem functions can match $A \subset \Gamma$. The following are the cases to consider:

1. R_2 may be an evaluation rule, so σA is one evaluation. In this case, R_1 can clearly match σA because substitutions only affect destinations and applicable evaluation rules are not picked based on the structure of destinations.
2. R_2 may be a trigger rule. For simplicity, assume that σA has the pattern `ret V X * cont F X W`. A could have the pattern `ret V X1 * cont F X2 W` and σ substituted the same destination for `X1` and `X2`.

   2.1. `X1 = X2` in Γ, so R_1 matches A.
   R_1 produces $\Gamma' \equiv (\Gamma, C)$ for some consequent C.
   R_2 produces $\Delta' \equiv (\Delta, \sigma'\sigma C)$, where σ' is the substitution for any newly generated existential variables in C that would have been skolemized. The remainder of the proof proceeds as it did for the soundness direction.

   2.2. `X1` and `X2` do not unify in Γ. For the remainder of the proof, let us refer to the continuations from A and σA respectively as K_1 and σK_1.

   2.2.1. One or both destinations may be atomic destinations generated in the initial state. The skolemization strategy does not substitute into atomic destinations, nor does it substitute atomic destinations for other variables because it consistently uses a destination constructors of the form `(d/x W)`. This means `X1` and `X2` would have remained unchanged in Δ, and R_2 could not have fired. This is a contradiction.

   2.2.2. Both destinations are generated by different existential quantifiers. Due to S3OS requirements, we know that there must exist a continuation $K_2 \in \Gamma$ waiting on `X1`. K_2 must have the pattern `cont F' X1 W'`.

   2.2.2.1. `F` and `F'` are constructed from different constructors or positionally nonequivalent terms. In this case, we have a contradiction of lemma A.1.3, because σK_1 and σK_2 from Δ would be two continuations with different frames waiting on the same destination.

   2.2.2.2. `F` and `F'` are constructed using the same constructor and contain positionally equivalent terms modulo substitution. In this case, we know that σK_2 has the pattern `cont F X W'`. Our skolemization strategy must have constructed `X` and `W'` from the same target destination when producing σK_2. It must have also produced `X` and `W` from the same target destination destination when producing σK_1. Therefore `W = W'`, i.e. σK_1 and σK_2 are equivalent. We can pretend that R_2 actually matched σK_2 in the antecedent σA, in which case R_1 would be able to match A. The proof proceeds as in case 2.1.

In case 2, if instead of a simple trigger rule, we had one like `tr/dest` which matched a returned value to a destination substituted into a source term, we would have to appeal to lemma A.1.2 instead of lemma A.1.3 in case 2.2.2.1.

Also note that the presentation of the trigger rules could easily be generalized to include `cont1`. Also backward chaining rules which compute results from terms occurring in the returns

and continuations will be applicable in both S3OS and S3OS-sko, because there is no loss of information in the value or valuable terms, and they are all equivalent when the resources have been matched up. □

A.1.2 Proof of Theorem 3.2 from Section 3.2.2

The following proves that S3OS-comp saturates.

Proof. We prove by establishing that every step taken during saturation converges towards a final state of the state after a finite number of steps.

Here, we define each step to be the application of all forward inference rules that match any resources that already exist in the current state Γ to grow it to $\Gamma \cup \Gamma'$. Since we have skolemized or erased arbitrary variables generated by existential quantifiers, we know that each rule can match any set of resources only once. Also, since we are working with a defunctionalized semantics, there can only be a finite number of rules. So a step, as we have defined here is a finite, terminating change to the state. The next step can only match resources in $\Gamma' \backslash \Gamma$.

The convergence requirement is that each step produces evaluations and continuations constructed from smaller terms than those already in Γ. We ignore returns in this convergence criterion because we only have evaluation and trigger rules. Evaluation rules can't fire without evaluations, and trigger rules cannot fire without continuations. To prove this convergence requirement, it is sufficient to show that every forward inference rule produces smaller evaluations and continuations than those matched in the antecedent. This is an S3OS requirement.

We illustrate with a few representative cases from $\mathcal{L}$:

- Consider `ev/op`. The antecedent is an evaluation over `(exp/op E1 B E2)`. The consequent produces:
  - An evaluation over E1, which is smaller than the antecedent.
  - A continuation over `(f/op1 B E2)`. The frame contains smaller terms than those in the antecedent.
- Consider `tr/let`. The antecedent matches a continuation over `(f/let (\!d.  S !d))`. The consequent produces a continuation over `(S !(d/7 W))`. This is not a syntactic substructure of the antecedent, but is still smaller due to the removal of the binder.

□

A.1.3 Proof of Theorem 3.3 from Section 3.2.2

The key to this proof is the generalization property:

Definition A.1. *Δ is considered a generalization of Γ if and only if:*

- *The evaluations from Γ when collapsed into a set are a subset of the persistent evaluations in Δ*
- *The continuations from Γ when collapsed into a set are a subset of the persistent continuations in Δ*

- *The set of destinations at which Γ has returns is a subset of the set of destinations at which Δ has returns.*

Lemma A.1.4. *Suppose that Δ is a context produced by saturation the S3OS-comp specification on a program. Then Δ is a generalization of any state reachable by the S3OS-sko specification.*

Proof. We know that Δ is a generalization of the initial state of the S3OS-sko specification.

Assume that Δ is a generalization of some arbitrary intermediate state Γ reached by S3OS-sko (inductive hypothesis). Suppose an arbitrary transition $\Gamma \to \Gamma'$ occurs such that Δ is not a generalization of Γ'. This means that a rule R_1 from S3OS-sko fired on Γ such that corresponding R_2 after erasure from S3OS-comp could not fire on Δ. By case analysis on each rule in S3OS-comp, we know that this is impossible. □

The proof that the S3OS-comp specification collects a superset of continuations encountered by the corresponding S3OS-sko specification now follows quite simply, because Δ is a generalization of any set of continuations collected by any arbitrary execution of the program by S3OS-sko.

A.1.4 Proof of Theorem 3.4 from Section 3.3

The following establishes bisimulation between S3OS-sko and S3OS-JIT:

Proof. The queries that load the initial state of Γ_0 and Δ_0 respectively ensure that Δ_0 is a compiling instantiation of Γ_0.

Suppose after n steps that the S3OS-sko computation is currently in a state Γ, and the S3OS-JIT computation is currently in a state Δ such that Δ is a compiling instantiation of Γ.

We show soundness. Suppose a rule R_1 causes a transition $\Gamma \to \Gamma'$. The S3OS-JIT version of the same rule (call it R_2) is also guaranteed to match Δ, because Δ contains the resources that match the same antecedent, and any continuations matched in the antecedent are persistent as required. R_2 causes a transition $\Delta \to \Delta'$. R_1 removes any matched evaluations, continuations and returns from Γ, and adds the resources from the consequent into Γ'. R_2 consumes evaluations and returns from Δ but leaves continuations untouched. It adds the same evaluations and returns to Δ' as R_1, and adds a corresponding persistent continuation only if it did not already exist in Δ. Therefore, Δ' is a compiling instantiation of Γ'.

Now we show completeness. Suppose a rule R_2 causes a transition $\Delta \to \Delta'$. We need to show that the S3OS-sko version of R_2 (call it R_1) can cause a transition $\Gamma \to \Gamma'$. There are two cases:

1. R_2 is an evaluation rule, in which case we know that the matched evaluation is also present in Γ, and R_1 can match Γ. In this case we can complete the proof as we did with soundness that Δ' is a compiling instantiation of Γ'.
2. R_2 is a trigger rule. The returned values matched in Δ are also guaranteed to be present in Γ.

   2.1. The continuation matched by R_2 in Δ occurs in Γ also, in which case R_1 can match Γ and the proof can be completed as in case 1.

2.2. The continuation matched by R_2 in Δ does not occur in Γ. For simplicity, assume that the antecedent matched by R_2 has the form `ret V X * !cont F X W`, and let the continuation be called K_2. There are two possibilities to consider:

2.2.1. There is no continuation in Γ that waits on X. This case cannot occur because we imposed an S3OS guideline requiring that every rule that produces a continuation must simultaneously produce the computation which triggers it. There should be an equal number of resources at X as there are continuations waiting on X in Γ.

2.2.2. There is a continuation K_1 in Γ of the form `cont F' X W'`, i.e. K_1 is also in Δ. We know from S3OS-JIT that Δ only accumulates continuations that have at some point been generated by the S3OS-sko specification, so we have the following cases:

2.2.2.1. `F` and `F'` are positionally nonequivalent. This case cannot occur because it would be a contradiction of lemma A.1.3. The key observation is that lemma A.1.3 is a property proven for all continuations ever produced during the lifetime of a program, and does not depend on rules consuming continuations.

2.2.2.2. `F` and `F'` are positionally equivalent. We know from our skolemization strategy (since K_1 and K_2 both wait on `X`) that they must have both been placed there by a rule matching positionally equivalent nonterminals. This means that `W = W'`. This contradicts the assumption made at the beginning of case 2.2 that the continuation matched by R_2 does not exist in Γ.

We assumed a simple antecedent for the trigger rule. If we had matched any returned values to destinations substituted within valuable terms, we would have had to make appropriate appeal to lemma A.1.2.

As in theorem 3.1, the proof cases for trigger rules can be easily generalized to include `cont1` and backward chaining rules. These cases have been omitted for brevity.

□

A.1.5 Proof of Theorem 3.5 from Section 3.3

Lemma A.1.5. *Suppose a S3OS-JIT specification is evaluating a program and is at some state Γ. If all of the continuations found in the consequent of an evaluation rule R are present in Γ, then R must have matched some previous state in the trace.*

We do not give a formal proof of this lemma. Intuitively, it follows from the use of distinct destination constructors in each rule. The constructed destinations used in a set of continuations uniquely identify the rule which must have produced them.

The following establishes bisimulation between S3OS-JIT and S3OS-VM:

Proof. The queries that load the initial state Γ_0 and Δ_0 respectively ensure that Δ_0 is an ahead-of-time instantiation of Γ_0. Furthermore, Δ_0 contains every continuation that will every be produced by Γ_0 (theorem 3.3).

Suppose after n steps that the S3OS-JIT computation is currently in a state Γ, and the S3OS-VM computation is currently in a state Δ such that Δ is an ahead-of-time instantiation of of Γ.

We show soundness. Suppose a rule R_1 causes a transition $\Gamma \rightarrow \Gamma'$. We wish to show that the S3OS-VM version of the same rule (call it R_2) can cause the transition $\Delta \rightarrow \Delta'$ such that Δ' is an ahead-of-time instantiation of Γ'. There are two cases:

1. R_1 is a trigger rule, in which case it is obvious that R_2 can fire. Δ' is an ahead-of-time instantiation of Γ'. because the rules made the same changes to evaluations and returns, and because Δ' already contained all the continuations that could every be generated by Γ.
2. R_1 is an evaluation rule, in which case it produces a nonterminal continuation at the same destination as the antecedent. We know that all continuations added into Γ' were already in Δ, so R_2 matches Δ. Is in (case 1), we complete the proof that Δ' is an ahead-of-time instantiation of Γ'

Now we show completeness. Suppose a rule R_2 causes a transition $\Delta \rightarrow \Delta'$. We need to show that the S3OS-JIT version of R_2 (call it R_1) can cause a transition $\Gamma \rightarrow \Gamma'$. There are two cases:

1. R_2 is a trigger rule, in which case the returned values matched by R_2 are also present in Γ. This case mirrors case 2 of completeness from the bisimulation of S3OS-sko and S3OS-JIT, so we do not restate it here.
2. R_2 is an evaluation rule which matched an evaluation at some destination `W`. An evaluation at the destination `W` (call it E_1) must also be present in Γ, and we need to show that this evaluation is matched by R_1. R_2 also matches on continuations in the antecedent (which have constructed destinations). Suppose these continuations were present in some state Γ' that occurs after Γ in Θ. Then we could appeal to lemma A.1.5 on those continuations to show that an evaluation matching R_1 must have been present in Γ or some previous state in the trace Θ. Due to lemma A.1.1, E_1 has to be this evaluation and R_1 does indeed match Γ. The remainder of the proof is completed as in the proof if soundness.

□

A.2 Properties from Chapter 4

A.2.1 Proof of Theorem 4.1 from Section 4.2

Lemma A.2.1. *Any two nonterminals are placed at different destinations by a skolemized S4OS specification, unless they contain the same frame and positionally equivalent source terms.*

Proof. Requirement 4.1 ensures that each source entry point used to seed the state is placed at a unique destination. Therefore, we can never construct the same destination for nonterminals produced from different source entry points. We simply need to show that nonterminals produced from one source entry point still satisfy the theorem. The proof is identical to that of lemma A.1.1. □

Lemma A.2.2. *Any two continuations produced by a skolemized S4OS specification must wait on different destinations, unless the continuations contain the same frame and positionally equivalent source terms.*

Proof. As in lemma A.2.1, we know that destinations constructed from different source entry points must be different. We simply need to prove the property for continuations that are produced from one source entry point. The structure of the proof is identical to that of lemma A.1.3, but appeals to lemma A.2.1 instead of lemma A.1.1. □

The following establishes bisimulation between S4OS and S4OS-sko (theorem 4.1):

Proof. This proof is very similar to that of theorem 3.1, therefore we do not present it. For both soundness and completeness, there is an identical proof that the same rule is applicable to Γ and Δ respectively. If the rule produced no new closure, nothing interesting happens with κ, and the mapping of destinations does not have to be updated. If the rule produces a new closure for Γ' and Δ', we simply extend κ with a one-to-one mapping between the new closures.

The proof is also simplified compared to theorem 3.1 because there are no longer any lambda expressions and substitutions to consider. □

A.2.2 Updates due to Loops in Section 4.3

Lemma A.2.3. *Revision of lemma A.2.1 to accommodate size metric from definition 4.4 and requirement 4.6:*

Any two nonterminals are placed at different destinations, unless they contain frames of the same size *and positionally identical terms.*

The proof is unchanged because we take care to use a distinct set of destination constructors in each rule (even those handling frames of the same size). The proof of lemma A.2.2 is unaffected for a similar reason.

Appendix B

$\mathcal{L}$ listings for chapter 3

B.1 Support Code

This is the part of the $\mathcal{L}$ specification that does not change at every stage of compiler and VM synthesis. We have also refactored the skolemized specification, compiler and VM so that the specification of simplified instructions and destination constructors are listed in one file only.

```
bit : type.    b0    : bit.    b1    : bit.
bin : type.    bin/e : bin.    bin/b : bin -> bit -> bin.

bin/add : bin -> bin -> bit -> bin -> bit -> type. #mode bin/add + + + - -.
bit/add : bit -> bit -> bit -> bit -> bit -> type. #mode bit/add + + + - -.

bit/add/000 : bit/add b0 b0 b0 b0 b0.  bit/add/001 : bit/add b0 b0 b1 b0 b1.
bit/add/010 : bit/add b0 b1 b0 b0 b1.  bit/add/100 : bit/add b1 b0 b0 b0 b1.
bit/add/011 : bit/add b0 b1 b1 b1 b0.  bit/add/101 : bit/add b1 b0 b1 b1 b0.
bit/add/110 : bit/add b1 b1 b0 b1 b0.  bit/add/111 : bit/add b1 b1 b1 b1 b1.

bin/add/bit : bin/add (bin/b N1 B1) (bin/b N2 B2) C (bin/b N3 B3) C''
               o- bit/add C B1 B2 C' B3
               o- bin/add N1 N2 C' N3 C''.

bin/add/nil : bin/add bin/e bin/e C bin/e C.

dest   : type.
op     : type.  op/+    : op.
exp    : type.  exp/num : bin -> exp.  exp/dest : dest -> exp.
exp/op : exp -> op -> exp -> exp.

stm : type. stm/return : exp -> stm.  stm/let : exp -> (dest -> stm) -> stm.

valuable : type.  vbl/e : exp -> valuable.  vbl/s : stm -> valuable.
value    : type.  val/e : exp -> value.     val/dummy : value.

d/1   : dest -> dest.  d/2 : dest -> dest.  d/3 : dest -> dest.
d/4   : dest -> dest.  d/5 : dest -> dest.  d/6 : dest -> dest.
d/7   : dest -> dest.  d/8 : dest -> dest.  d/9 : dest -> dest.
```

```
frame    : type.                        inst     : type.
f/op1    : op -> valuable -> frame.     i/op1    : inst.
f/op2    : op -> frame.                 i/op2    : op -> inst.
f/let    : (dest -> stm) -> frame.      i/let    : inst.
f/dummy  : frame.                       i/dummy  : inst.
f/num    : bin -> frame.                i/num    : bin -> inst.
f/dest   : dest -> frame.               i/dest   : dest -> inst.

cont  : frame -> dest -> dest -> type.          #mode cont  - - -.
cont1 : dest -> frame -> dest -> dest -> type.  #mode cont1 - - - -.

comp  : inst  -> dest -> dest -> type.          #mode comp  - - -.
comp1 : dest -> inst  -> dest -> dest -> type.  #mode comp1 - - - -.
```

B.2 S3OS-sko

This is the S3OS specification for $\mathcal{L}$ with skolemized destinations.

```
eval : valuable -> dest -> type.  #mode eval - -.
ret  : value -> dest -> type.     #mode ret  - -.

ev/num : eval (vbl/e (exp/num N)) W
         -o { ret val/dummy (d/1 W) * cont (f/num N) (d/1 W) W }.

tr/num : ret val/dummy X * cont (f/num N) X W
         -o { ret (val/e (exp/num N)) W }.

tr/dummy : ret V X * cont f/dummy X W -o { ret V W }.

ev/op : eval (vbl/e (exp/op E1 B E2)) W
        -o { eval (vbl/e E1) (d/2 W) *
             cont (f/op1 B (vbl/e E2)) (d/2 W) W }.

tr/op1 : ret V1 X * cont (f/op1 B E2) X W
         -o { ret V1 (d/3 W) * eval E2 (d/4 W) *
              cont1 (d/3 W) (f/op2 B) (d/4 W) W }.

tr/op2/add : ret (val/e (exp/num N1)) X1 *
             ret (val/e (exp/num N2)) X2 *
             cont1 X1 (f/op2 op/+) X2 W *
             !bin/add N1 N2 b0 N3 _
             -o { ret (val/e (exp/num N3)) W }.

ev/return : eval (vbl/s (stm/return E)) W -o { eval (vbl/e E) W }.

ev/dest : eval (vbl/e (exp/dest D)) W
          -o { ret val/dummy (d/5 W) * cont (f/dest D) (d/5 W) W }.

tr/dest : ret val/dummy X * cont (f/dest D) X W * !ret V D -o { ret V W }.

ev/let : eval (vbl/s (stm/let E S)) W
```

```
         -o { eval (vbl/e E) (d/6 W) * cont (f/let S) (d/6 W) (d/8 W) *
              cont f/dummy (d/8 W) W }.

tr/let : ret V X * cont (f/let (\!d. S !d)) X W
         -o { !ret V (d/7 W) * eval (vbl/s (S !(d/7 W))) (d/9 W) *
              cont f/dummy (d/9 W) W }.
```

B.3 S3OS-comp

Here, we list S3OS-comp and the code simplifier together because both pieces of software work together. However, they were not both synthesized at the same time.

```
eval  : valuable -> dest -> type.  #mode eval  - -.
ret   : dest -> type.              #mode ret   -.

ev/num : !eval (vbl/e (exp/num N)) W
         -o { !ret (d/1 W) * !cont (f/num N) (d/1 W) W }.

tr/num : !ret X * !cont (f/num N) X W -o { !ret W }.

tr/dummy : !ret X * !cont f/dummy X W -o { !ret W }.

ev/op : !eval (vbl/e (exp/op E1 B E2)) W
        -o { !eval (vbl/e E1) (d/2 W) *
             !cont (f/op1 B (vbl/e E2)) (d/2 W) W }.

tr/op1 : !ret X * !cont (f/op1 B E2) X W
         -o { !ret (d/3 W) * !eval E2 (d/4 W) *
              !cont1 (d/3 W) (f/op2 B) (d/4 W) W }.

tr/op2/add : !ret X1 * !ret X2 * !cont1 X1 (f/op2 op/+) X2 W -o { !ret W }.

ev/return : !eval (vbl/s (stm/return E)) W -o { !eval (vbl/e E) W }.

ev/dest : !eval (vbl/e (exp/dest D)) W
          -o { !ret (d/5 W) * !cont (f/dest D) (d/5 W) W }.

tr/dest : !ret X * !cont (f/dest D) X W * !ret D -o { !ret W }.

ev/let : !eval (vbl/s (stm/let E S)) W
         -o { !eval (vbl/e E) (d/6 W) *
              !cont (f/let S) (d/6 W) (d/8 W) * !cont f/dummy (d/8 W) W }.

tr/let : !ret X * !cont (f/let (\!d. S !d)) X W
         -o { !ret (d/7 W) * !eval (vbl/s (S !(d/7 W))) (d/9 W) *
              !cont f/dummy (d/9 W) W }.

tr/frame: frame -> inst -> type. #mode tr/frame + -.
tr/frame/op1    : tr/frame (f/op1 B E) (i/op1).
tr/frame/op2    : tr/frame (f/op2 B)   (i/op2 B).
tr/frame/let    : tr/frame (f/let S)   (i/let).
```

```
tr/frame/dummy : tr/frame (f/dummy)   (i/dummy).
tr/frame/num   : tr/frame (f/num N)   (i/num N).
tr/frame/dest  : tr/frame (f/dest D)  (i/dest D).

tr/cont  : !cont F X W * tr/frame F I -o { !comp I X W }.
tr/cont1 : !cont1 X1 F X2 W * tr/frame F I -o { !comp1 X1 I X2 W }.
```

B.4 Simplified S3OS-VM

This is the virtual machine for the simplified instruction set output by the compiler in the previous section.

```
eval  : dest -> type.          #mode eval -.
ret   : value -> dest -> type.  #mode ret  - -.

ev/num : eval W * !comp (i/num N) (d/1 W) W -o { ret val/dummy (d/1 W) }.

tr/num : ret val/dummy X * !comp (i/num N) X W
         -o { ret (val/e (exp/num N)) W }.

tr/dummy : ret V X * !comp i/dummy X W -o { ret V W }.

ev/op : eval W * !comp i/op1 (d/2 W) W -o { eval (d/2 W) }.

tr/op1 :
 ret V1 X * !comp i/op1 X W
 -o { ret V1 (d/3 W) * eval (d/4 W) }.

tr/op2/add : ret (val/e (exp/num N1)) X1 * ret (val/e (exp/num N2)) X2 *
             !comp1 X1 (i/op2 op/+) X2 W * !bin/add N1 N2 b0 N3 _
             -o { ret (val/e (exp/num N3)) W }.

ev/dest : eval W * !comp (i/dest D) (d/5 W) W -o { ret val/dummy (d/5 W) }.

tr/dest : ret val/dummy X * !comp (i/dest D) X W * ret V D -o { ret V W }.

ev/let : eval W * !comp i/let (d/6 W) (d/8 W) * !comp i/dummy (d/8 W) W
         -o { eval (d/6 W) }.

tr/let : ret (val/e V) X * !comp i/let X W
         -o { !ret (val/e V) (d/7 W) * eval (d/9 W) }.
```

Appendix C

Higher Order Features

This chapter is included for the advanced reader to demonstrate how one can encode higher order and functional programming features according to the rules of S4OS. The key idea is to reuse the explicit closure to represent the closure structure of the language. We elucidate using the untyped call-by-name lambda calculus. Futures are also presented, to illustrate a concurrent feature.

C.1 De Bruijn Indices and Closures

This section defines the abstract syntax of expressions and lists the support code handling variables. Variables are explicitly encoded as natural numbers (celfnat) corresponding to de Bruijn indices. The support code explains how these indices are used to look up the explicit closure for a value bound by the correct λ binder.

```
nat     : type.  nat/z  : nat.    nat/s : nat -> nat.
exp     : type.
exp/idx : nat -> exp.  exp/lam : exp -> exp.  exp/app : exp -> exp -> exp.

nat-lt   : nat -> nat -> type. #mode nat-lt + +.
nat-lt/z : nat-lt nat/z (nat/s N2).
nat-lt/s : nat-lt (nat/s N1) (nat/s N2) o- nat-lt N1 N2.

nat-gte   : nat -> nat -> type. #mode nat-gte + +.
nat-gte/z : nat-gte N1 nat/z.
nat-gte/s : nat-gte (nat/s N1) (nat/s N2) o- nat-gte N1 N2.

clo     : type.  clo/cons : clo -> clo -> type. #mode clo/cons - +.

lookup   : clo -> nat -> clo -> type. #mode lookup + + -.
lookup/z : lookup C nat/z C.
lookup/s : lookup C (nat/s N) C'' o- clo/cons C' C * lookup C' N C''.

exp-shift : exp -> nat -> exp -> type. #mode exp-shift + + -.

exp-shift/idx/free : exp-shift (exp/idx N) FV (exp/idx (nat/s N))
                     o- nat-gte N FV.
```

```
exp-shift/idx/bound : exp-shift (exp/idx N) FV (exp/idx N)
                       o- nat-lt N FV.

exp-shift/lam : exp-shift (exp/lam E) FV (exp/lam E')
                 o- exp-shift E (nat/s FV) E'.

exp-shift/app : exp-shift (exp/app E1 E2) FV (exp/app E1' E2')
                 o- exp-shift E1 FV E1'
                 o- exp-shift E2 FV E2'.
```

`exp-shift` is used to increment the indices corresponding to free variables in an expression This can be used if we need to place the term inside another lambda without changing the meaning of the variables.

C.2 Call by name λ Calculus

```
runtime  : type.
dest     : type.
dest/1   : dest.

valuable : type.  vbl/e  : exp -> valuable.
value    : type.  val/d  : value.  val/r : runtime -> value.

r/clo-val : dest -> clo -> runtime.  r/wait : clo -> runtime.

frame0/1 : type.              frame1/1 : type.
f/idx    : nat -> frame0/1.   f/lam    : exp -> frame0/1.
f/dummy  : frame0/1.          f/wait   : frame1/1.

eval    : valuable -> clo -> dest -> type.          #mode eval - - -.
ret     : value -> clo -> dest -> type.             #mode ret - - -.
cont0/1 : frame0/1 -> dest -> dest -> type.         #mode cont0/1 - - -.
cont1/1 : dest -> frame1/1 -> dest -> dest -> type. #mode cont1/1 - - - -.
src     : valuable -> dest -> type.                 #mode src - -.

eval/idx :
  eval (vbl/e (exp/idx I)) C W
  -o { Exists x. ret val/d C x * cont0/1 (f/idx I) x W }.

eval/lam :
  eval (vbl/e (exp/lam E)) C W
  -o { Exists x. ret val/d C x * cont0/1 (f/lam E) x W }.

trigger/lam :
  ret val/d C X * cont0/1 (f/lam E) X W
  -o { Exists l. !src (vbl/e E) l * ret (val/r (r/clo-val l C)) C W }.

trigger/wait :
  ret V C2 X1 * ret (val/r (r/wait C2)) C1 X2 *
  cont1/1 X2 f/wait X1 W
  -o { ret V C1 W }.
```

The above listing specifies those parts of the semantics that call by name and futures have in common. Principal amongst the shared features is the evaluation of lambda expressions, which are first class values. In S4OS, static and dynamic terms must be kept separate. Therefore it is not permitted to return source expressions as values, whether or not we use substitution. Instead, the text of the lambda expression is placed in a source entry point. The returned value must indirectly refer to the source entry point and the closure in which it may be entered.

Unlike in $\mathcal{L}_{\mathsf{fun}}$, this specification creates source entry points during evaluation. We can still guarantee that they satisfy BSV because source entry points are only created using terms that are proper syntactic substructure of the original program text. Also unlike in $\mathcal{L}_{\mathsf{fun}}$, this specification refers to closures from values. This is acceptable because closures are considered dynamic. In order to move values from one closure to another, we introduce `f/wait`.

C.2.1 Application

Here is a plausible implementation of application:

```
f/app      : exp -> frame0/1.
r/clo-exp : dest -> clo -> runtime.

eval/app-name :
  eval (vbl/e (exp/app E1 E2)) C W
  -o { Exists x1. eval (vbl/e E1) C x1 *
       Exists x2. cont0/1 (f/app E2) x1 x2 * cont0/1 f/dummy x2 W }.

trigger/app-name :
  ret (val/r (r/clo-val L C2)) C1 X1 * cont0/1 (f/app E2) X1 W * !src E L
  -o { Exists l. !src (vbl/e E2) l * Exists c3. !clo/cons c3 C2 *
       !ret (val/r (r/clo-exp l C1)) c3 dest/1 *
       Exists x3. eval E c3 x3 * Exists x4. ret (val/r (r/wait c3)) C1 x4 *
       cont1/1 x4 f/wait x3 W }.

trigger/idx/exp :
  ret val/d C1 X * cont0/1 (f/idx I) X W * !lookup C1 I C2 *
  !ret (val/r (r/clo-exp L C3)) C2 dest/1 * !src E L
  -o { Exists x1. eval E C3 x1 *
       Exists x2. ret (val/r (r/wait C3)) C1 x2 * cont1/1 x2 f/wait x1 W }.
```

The above code may be a correct interpreter, but is unsuitable for synthesizing into S4OS-sko and S4OS-comp. We require that every evaluation of a source entry point must occur in a fresh closure so that the destination may always be skolemized based on the destination of the source entry point. In call by name, the argument may be evaluated multiple times at the same closure. The following implementation solves the problem by shifting the expression and requiring any attempt to evaluate the expression to bind an unused value so that the closure may have the correct structure. The newly bound argument forces the creation of a fresh explicit closure.

```
f/app      : exp -> frame0/1.
r/clo-exp : dest -> clo -> runtime.

eval/app-name :
  eval (vbl/e (exp/app E1 E2)) C W
  -o { Exists x1. eval (vbl/e E1) C x1 *
```

```
      Exists x2. cont0/1 (f/app E2) x1 x2 * cont0/1 f/dummy x2 W }.

trigger/app-name :
  ret (val/r (r/clo-val L C2)) C1 X1 * cont0/1 (f/app E2) X1 W *
  !src E L * !exp-shift E2 nat/z E2'
  -o { Exists l. !src (vbl/e E2') l * Exists c3. !clo/cons c3 C2 *
       !ret (val/r (r/clo-exp l C1)) c3 dest/1 *
       Exists x3. eval E c3 x3 * Exists x4. ret (val/r (r/wait c3)) C1 x4 *
       cont1/1 x4 f/wait x3 W }.

trigger/idx/exp :
  ret val/d C1 X * cont0/1 (f/idx I) X W * !lookup C1 I C2 *
  !ret (val/r (r/clo-exp L C3)) C2 dest/1 * !src E L
  -o { Exists c4. clo/cons c4 C3 * Exists x1. eval E c4 x1 *
       Exists x2. ret (val/r (r/wait c4)) C1 x2 * cont1/1 x2 f/wait x1 W }.
```

This technique is similar to a popularly used thunking strategy used in call by value languages to mimic lazy computation. The lazy computation is embedded in a function that takes a unit argument. Since we know that the bound value will never be used, we simply create a fresh closure at which to evaluate the expression without binding a value at the closure.

C.3 Futures

Futures allow the body of a called function and its argument to evaluate asynchronously. If the function requires the value of the argument before the argument finishes evaluation, the function waits. The concurrent computations are evaluated in separate closures.

```
frame1/2  : type.  f/wait2   : frame1/2.
f/future  : exp -> frame0/1.
f/promise : frame1/1.
r/promise : clo -> runtime.

cont1/2 : dest -> frame1/2 -> dest -> dest -> dest -> type.
#mode cont1/2 - - - - -.

eval/future :
  eval (vbl/e (exp/app E1 E2)) C W
  -o { Exists x1. eval (vbl/e E1) C x1 *
       Exists x2. cont0/1 (f/future E2) x1 x2 * cont0/1 f/dummy x2 W }.

trigger/future :
  ret (val/r (r/clo-val L C2)) C1 X1 *
  cont0/1 (f/future E2) X1 W * !src E L
  -o { Exists c3. !clo/cons c3 C2 * Exists x1. eval (vbl/e E2) C1 x1 *
       Exists x2. ret (val/r (r/promise C1)) c3 x2 *
       Exists x5. cont1/1 x2 f/promise x1 x5 *
       Exists x3. eval E c3 x3 * Exists x4. ret (val/r (r/wait c3)) C1 x4 *
       cont1/2 x4 f/wait2 x3 x5 W }.

trigger/promise :
  ret V C1 X1 * ret (val/r (r/promise C1)) C3 X2 *
```

```
  cont1/1 X2 f/promise X1 W
  -o { !ret V C3 dest/1 * ret V C1 W }.

trigger/wait2 :
  ret V3 C3 X3 * ret (val/r (r/wait C3)) C1 X4 *
  ret V5 C1 X5 * cont1/2 X4 f/wait2 X3 X5 W
  -o { ret V3 C1 W }.
```

We were able to successfully synthesize a virtual instruction set using our techniques on call by name and futures.

Bibliography

[ABDM03a] M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. *From interpreter to compiler and virtual machine: a functional derivation*. BRICS, Department of Computer Science, University of Aarhus, 2003. 5.4

[ABDM03b] M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19. ACM, 2003. 5.4

[AP93] K.R. Apt and D. Pedreschi. Reasoning about termination of pure prolog programs. *Information and computation*, 106:109–109, 1993. 1

[Cer] I. Cervesato. Personal correspondence. 5.3

[CPWW03] I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework ii: Examples and applications. Technical report, DTIC Document, 2003. 1, 2.1, 3, 2.1.2, 5.3

[CS09] I. Cervesato and A. Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044–1077, 2009. 1, 2.1.2

[Dav96] R. Davies. A temporal-logic approach to binding-time analysis. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 184–195. IEEE, 1996. 2.2

[DV96] O. Danvy and R. Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In *Programming Languages: Implementations, Logics, and Programs*, pages 182–197. Springer, 1996. 1, 5.4

[Fut99] Y. Futamura. Partial evaluation of computation processan approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. 2.2

[GL00] L. George and A. Leung. Mlrisc: A framework for retargetable and optimizing compiler backends. *Availble at http://cs1. cs. nyu. edu/leunga/www/MLRISC/-Doc/html*, 2000. 5.1

[GM02] H. Ganzinger and D. McAllester. Logical algorithms. *Logic Programming*, pages 31–42, 2002. 3.2

[GR68] C.C. Green and B. Raphael. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 1968 23rd ACM national conference*,

pages 169–181. ACM, 1968. 3.2

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993. 2.1.1

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta informatica*, 11(1):31–55, 1978. 2.1.1

[Jon96] N. Jones. What not to do when writing an interpreter for specialisation. *Partial evaluation*, pages 216–237, 1996. 2.2, 2.2

[Jør91] J. Jørgensen. Compiler generation by partial evaluation. *Master's thesis, DIKU, University of Copenhagen*, 1991. 5.4

[JS80] N. Jones and D. Schmidt. Compiler generation from denotational semantics. *Semantics-Directed Compiler Generation*, pages 70–93, 1980. 5.4

[Kah87] G. Kahn. Natural semantics. *STACS 87*, pages 22–39, 1987. 1

[Lat02] C.A. Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois, 2002. 5.1

[Lee87] P. Lee. *The automatic generation of realistic compilers from high-level semantic descriptions*. PhD thesis, University of Michigan, 1987. 1, 5.4

[Min96] J. Minker. Logic and databases: A 20 year retrospective. *Logic in Databases*, pages 1–57, 1996. 3.2

[Mos96] P. Mosses. Theory and practice of action semantics. *Mathematical Foundations of Computer Science 1996*, pages 37–61, 1996. 1, 5.4

[Mos99] P. Mosses. Foundations of modular sos. *Mathematical Foundations of Computer Science 1999*, pages 70–80, 1999. 1

[Ørb94] P. Ørbæk. Oasis: An optimizing action-based compiler generator. In *Compiler Construction*, pages 1–15. Springer, 1994. 1, 5.4

[PD01] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical structures in computer science*, 11(04):511–540, 2001. 2.2

[PE88] F. Pfenning and C. Elliot. Higher-order abstract syntax. *ACM SIGPLAN Notices*, 23(7):199–208, 1988. 2.1.1

[Pfe04] F. Pfenning. Substructural operational semantics and linear destination-passing style. *Lecture Notes in Computer Science*, pages 196–196, 2004. 1, 2.1, 2.1.2, 5.3

[Pfe06] F. Pfenning. Substructural operational semantics. Lecture notes for 15-819K: Logic Programming, available online at `http://www.cs.cmu.edu/afs/cs/user/fp/www/courses/lp/lectures/25-ssos.pdf`, 2006. 1, 5.3

[Pfe11] F. Pfenning. C0 reference. C0 Language Reference, available online at `http://c0.typesafety.net/doc/c0-reference.pdf`, 2011. 1, 5

[Pfe12a] F. Pfenning. C0 virtual machine. 15122 Lecture Notes, available online at `http://www.andrew.cmu.edu/course/15-122/lectures/23-c0vm.pdf`, 2012. 5.2

[Pfe12b] F. Pfenning. Lecture notes on linear logic. Lecture notes for 15-816: Linear Logic, available online at `http://www.cs.cmu.edu/~fp/courses/15816-s12/`, 2012. 2.1, 3

[PS09] F. Pfenning and R.J. Simmons. Substructural operational semantics as ordered logic programming. In *Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on*, pages 101–110. IEEE, 2009. 1, 2.1, 2.1.2, 5.1, 5.3

[PuDa81] G.D. Plotkin and Aarhus universitet. Datalogisk afdeling. A structural approach to operational semantics. 1981. 1

[RP96] E. Rohwedder and F. Pfenning. Mode and termination checking for higher-order logic programs. *Programming Languages and Systems–ESOP'96*, pages 296–310, 1996. 1

[Sim12] R.J. Simmons. *Substructural Logical Specifications (fourth committee draft)*. PhD thesis, Carnegie Mellon University, 2012. 1, 2.1, 2.1.2, 5.3

[SP09] R.J. Simmons and F. Pfenning. Linear logical approximations. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 9–20. ACM, 2009. 1, 3.2.1

[SS71] D.S. Scott and C. Strachey. *Toward a mathematical semantics for computer languages*. Oxford University Computing Laboratory, Programming Research Group, 1971. 1

[War77] D.H.D. Warren. *Implementing Prolog: compiling predicate logic programs*. Department of Artificial Intelligence, University of Edinburgh, 1977. 1

[WCPW03] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework i: Judgments and properties. Technical report, DTIC Document, 2003. 1, 2.1, 3, 5.3