Chapter 1

Introduction

Suppose you find yourself in possession of

* a calculator of unfamiliar design, or
* a new board game, or
* the control system for an army of robots, or
* an implementation of a security protocol, or
* the interface to a high-frequency trading system.

The fundamental questions are the same: *What does it do? What are the rules of the game?* The answer to this question, whether it comes in the form of an instruction manual, a legal document, or an ISO standard, is a *specification*.

Specifications must be *formal*, because any room for misinterpretation could (respectively) lead to incorrect calculations, accusations of cheating, a robot uprising, a security breach, or bankruptcy. At the same time, specifications must be *clear*: while clarity is in the eye of the beholder, a specification that one finds hopelessly confusing or complex is no more useful than one that is hopelessly vague. Clarity is what allows us to communicate with each other, to use specifications to gain a common understanding of what some system does and to think about how that system might be changed. Formality is what allows specifications to interact with the world of computers, to say with confidence that the *implementation* of the calculator or high-frequency trading system obeys the specification. Formality also allows specifications to interact with the world of mathematics, and this, in turn, enables us to make precise and accurate statements about what may or may not happen to a given system.

The specification of many (too many!) critical systems still remains in the realm of English text, and the inevitable lack of formality can and does make formal reasoning about these specifications difficult or impossible. Notably, this is true about most of the programming languages used to implement our calculators, program our robot army control systems, enforce our security protocols, and interact with our high-frequency trading systems. In the last decade, however, we have finally begun to see the emergence of operational semantics specifications (the "rules of the game" for a programming language) for real-world programming languages that are truly formal. A notable aspect of this recent work is that the formalization effort is not done simply for formalization's sake. Ellison and Roşu's formal semantics of C can be used to check individual

programs for undefined behavior, unsafe situations where the rules of the game no longer apply and the compiler is free to do anything, including unleashing the robot army [ER12]. Lee, Crary, and Harper's formalization of Standard ML has been used to formally prove – using a computer to check all the proof's formal details – a much stronger safety property: that *every* program accepted by the compiler is free of undefined behavior [LCH07].

Mathematics, by contrast, has a century-long tradition of insisting on absolute formality (at least in principle: practice often falls far short). Over time, this tradition has become a collaboration between practicing mathematicians and practicing computer scientists, because while humans are reasonable judges of clarity, computers have absolutely superhuman patience when it comes to checking all the formal details of an argument. One aspect of this collaboration has been the development of *logical frameworks*. In a logical framework, the language of specifications is derived from the language of logic, which gives specifications in a logical framework an independent meaning based on the logic from which the logical framework was derived. To be clear, the language of logic is not a single, unified entity: logics are formal systems that satisfy certain internal coherence properties, and we study many of them. For example, the logical framework Coq is based on the Calculus of Inductive Constructions [Coq10], the logical framework Agda is based on a variant of Martin-Löf's type theory called UTT_{Σ} [Nor07], and the logical framework Twelf is based on the dependent type theory λ^{Π}, also known as LF [PS99]. Twelf was the basis of Lee, Crary, and Harper's formalization of Standard ML.

Why is there not a larger tradition of formally specifying the programming languages that people actually use? Part of the answer is that most languages that people actually use have lots of features – like mutable state, or exception handling, or synchronization and communication, or lazy evaluation – that are not particularly pleasant to specify using existing logical frameworks. Dealing with a few unpleasant features at a time might not be much trouble, but the combinations that appear in actual programming languages cause formal programming language specifications to be both unclear for humans to read and inconvenient for formal tools to manipulate. A more precise statement is that the addition of the aforementioned features is *non-modular*, because handling a new feature requires reconsidering and revising the rest of the specification. Some headway on this problem has been made by frameworks like the K semantic framework that are formal but not logically derived; the K semantic framework is based on a formal system of rewriting rules [RŞ10]. Ellison and Roşu's formalization of C was done in the K semantic framework.

This dissertation considers the specification of systems, particularly programming languages, in logical frameworks. We consider a particular family of logics, called *substructural logics*, in which logical propositions can be given an interpretation as rewriting rules as detailed by Cervesato and Scedrov [CS09]. We seek to support the following:

> **Thesis Statement:** *Logical frameworks based on a rewriting interpretation of substructural logics are suitable for modular specification of programming languages and formal reasoning about their properties.*[1]

Part I of the dissertation covers the design of logical frameworks that support this rewriting interpretation and the design of the logical framework SLS in particular. Part II considers the

[1] The original thesis proposal used the phrase "forward reasoning in substructural logics" instead of the phrase "a rewriting interpretation of substructural logics," but these are synonymous, as discussed in Section 4.6.

$$
\begin{array}{rl}
\mathsf{hd} < < > < < > > > & \rightsquigarrow \\
< \mathsf{hd} < > < < > > > & \rightsquigarrow \\
< < \mathsf{hd} > < < > > > & \rightsquigarrow \\
< \mathsf{hd} < < > > > & \rightsquigarrow \\
< < \mathsf{hd} < > > > & \rightsquigarrow \\
< < < \mathsf{hd} > > > & \rightsquigarrow \\
< < \mathsf{hd} > > & \rightsquigarrow \\
< \mathsf{hd} > & \rightsquigarrow \\
\mathsf{hd} &
\end{array}
$$

Figure 1.1: Series of PDA transitions

modular specification of programming language features in SLS and the methodology by which we organize and relate styles of specification. Part III discusses formal reasoning about properties of SLS specifications, with an emphasis on establishing invariants.

1.1 Logical frameworks

Many interesting stateful systems have a natural notion of *ordering* that is fundamental to their behavior. A very simple example is a push-down automaton (PDA) that reads a string of symbols left-to-right while maintaining and manipulating a separate stack of symbols. We can represent a PDA's internal configuration as a sequence with three regions:

$$[\text{ the stack }]\ [\text{ the head }]\ [\text{ the string being read }]$$

where the symbols closest to the head are the top of the stack and the symbol waiting to be read from the string. If we represent the head as a token hd, we can describe the behavior (the rules of the game) for the PDA that checks a string for correct nesting of angle braces by using two rewriting rules:

$$\mathsf{hd} < \rightsquigarrow < \mathsf{hd} \qquad \text{(push)}$$

$$< \mathsf{hd} > \rightsquigarrow \mathsf{hd} \qquad \text{(pop)}$$

The distinguishing feature of these rewriting rules is that they are *local* – they do not mention the entire stack or the entire string, just the relevant fragment at the beginning of the string and the top of the stack. Execution of the PDA on a particular string of tokens then consists of (1) appending the token hd to the beginning of the string, (2) repeatedly performing rewritings until no more rewrites are possible, and (3) checking to see if only a single token hd remains. One possible series of transitions that this rewriting system can take is shown in Figure 1.1

Because our goal is to use a framework that is both simple and logically motivated, we turn to a substructural logic called *ordered logic*, a fragment of which was originally proposed by

Lambek for applications in computational linguistics [Lam58]. In ordered logic, hypotheses are ordered relative to one another and cannot be rearranged. The rewriting rules we considered above can be expressed as propositions in ordered logic, where the tokens hd, $>$, and $<$ are all treated as *atomic propositions*:

$$\begin{aligned} \mathsf{push}: &\quad \mathsf{hd} \bullet < \;\rightarrowtail\; \{< \bullet \mathsf{hd}\} \\ \mathsf{pop}: &\quad < \bullet \mathsf{hd} \bullet > \;\rightarrowtail\; \{\mathsf{hd}\} \end{aligned}$$

The symbol $\bullet$ (pronounced "fuse") is the binary connective for ordered conjunction (i.e. concatenation); it binds more tightly than $\rightarrowtail$, a binary connective for ordered implication. The curly braces $\{\ldots\}$ can be ignored for now.

The propositional fragment of ordered logic is Turing complete: it is in fact a simple exercise to specify a Turing machine! Nevertheless, first-order quantification helps us write specifications that are short and clear. For example, by using first-order quantification we can describe a a more general push-down automaton in a generic way. In this generic specification, we use $\mathsf{left}(X)$ and $\mathsf{right}(X)$ to describe left and right angle braces ($X = \mathsf{an}$), square braces ($X = \mathsf{sq}$), and parentheses ($X = \mathsf{pa}$). The string [$<$ $>$ ([ ])] is then represented by the following sequence of ordered atomic propositions:

$$\mathsf{left}(\mathsf{sq})\ \ \mathsf{left}(\mathsf{an})\ \ \mathsf{right}(\mathsf{an})\ \ \mathsf{left}(\mathsf{pa})\ \ \mathsf{left}(\mathsf{sq})\ \ \mathsf{right}(\mathsf{sq})\ \ \mathsf{right}(\mathsf{pa})\ \ \mathsf{right}(\mathsf{sq})$$

The following rules describe the more general push-down automaton:

$$\begin{aligned} \mathsf{push}: &\quad \forall x.\, \mathsf{hd} \bullet \mathsf{left}(x) \;\rightarrowtail\; \{\mathsf{stack}(x) \bullet \mathsf{hd}\} \\ \mathsf{pop}: &\quad \forall x.\, \mathsf{stack}(x) \bullet \mathsf{hd} \bullet \mathsf{right}(x) \;\rightarrowtail\; \{\mathsf{hd}\} \end{aligned}$$

(This specification would still be possible in propositional ordered logic; we would just need one copy of the push rule and one copy of the pop rule for each pair of braces.) Note that while we use the fuse connective to indicate adjacent tokens in the rules above, no fuses appear in Figure 1.1. That is because the intermediate states are not propositions in the same way rules are propositions. Rather, the intermediate states in Figure 1.1 are *contexts* in ordered logic, which we will refer to as *process states*.

The most distinctive characteristic of these transition systems is that the intermediate stages of computation are encoded in the structure of a substructural context (a process state). This general idea dates back to Miller [Mil93] and his Ph.D. student Chirimar [Chi95], who encoded the intermediate states of a π-calculus and of a low-level RISC machine (respectively) as contexts in focused classical linear logic. Part I of this dissertation is concerned with the design of logical frameworks for specifying transition systems. In this respect, Part I follows in the footsteps of Miller's Forum [Mil96], Cervesato and Scedrov's multiset rewriting language ω [CS09], and Watkins et al.'s CLF [WCPW02].

As an extension to CLF, the logical framework we develop is able to specify systems like the π-calculus, security protocols, and Petri nets that can be encoded in CLF [CPWW02]. The addition of ordered logic allows us to easily incorporate specifications that are naturally expressed as string rewriting systems. An example from the verification domain, taken from Bouajjani and Esparza [BE06], is shown in Figure 1.2. The left-hand side of the figure is a simple Boolean

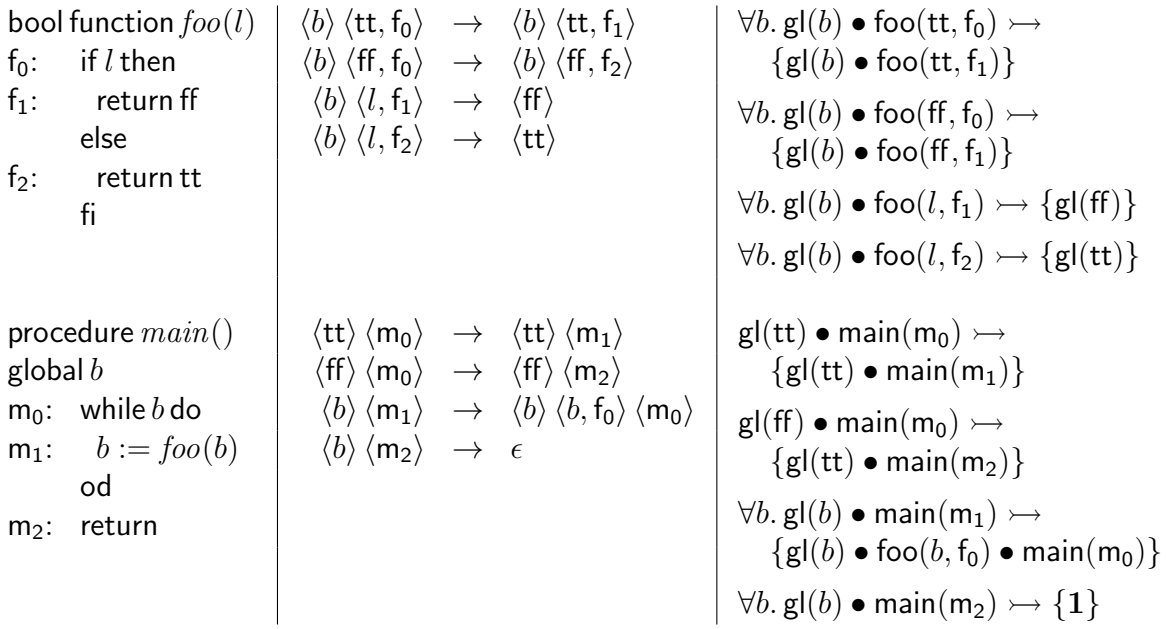

Figure 1.2: A Boolean program, encoded as a rewriting system and in SLS

program: the procedure foo has one local variable and the procedure $main$ has no local variables but mentions a global variable b. Bouajjani and Esparza represented Boolean programs like this one as *canonical systems* like the one shown in the middle of Figure 1.2. Canonical systems are rewriting systems where only the left-most tokens are ever rewritten: the left-most token in this canonical system always has the form $\langle b\rangle$, where b is either true (tt) or false (ff), representing the valuation of the global variables – there is only one, b. The token to the right of the global variables contains the current program counter and the value of the current local variables. The token to the right of *that* contains the program counter and local variables of the calling procedure, and so on, forming a call stack that grows off to the right (in contrast to the PDA's stack, which grew off to the left). Canonical systems can be directly represented in ordered logic, as shown on the right-hand side of Figure 1.2. The atomic proposition $\mathsf{gl}(b)$ contains the global variables (versus $\langle b\rangle$ in the middle column), the atomic proposition $\mathsf{foo}(l, f)$ contains the local variables and program counter within the procedure foo (versus $\langle l, f\rangle$ in the middle column), and the atomic proposition $\mathsf{main}(m)$ contains the program counter within the procedure main (versus $\langle m\rangle$ in the middle column).

The development of SLS, a CLF-like framework of **s**ubstructural **l**ogical **s**pecifications that includes an intrinsic notion of order, is a significant development of Part I of the dissertation. However, the principal contribution of these three chapters is the development of *structural focalization*, which unifies Andreoli's work on focused logics [And92] with the *hereditary substitution* technique that Watkins developed in the context of CLF [WCPW02]. Chapter 2 explains structural focalization in the context of linear logic, Chapter 3 establishes focalization for a richer substructural logic OL_3, and Chapter 4 takes focused OL_3 and carves out the SLS framework as a fragment of the focused logic.

1.2 Substructural operational semantics

Existing logical frameworks are perfectly capable of representing simple systems like PDAs, and while applications in the verification domain like the rewriting semantics of Boolean programs are an interesting application of SLS, they will not be a focus of this dissertation. Instead, in Part II, we will concentrate on specifying the operational semantics of programming languages in SLS. We can represent operational semantics in SLS in many ways, but we are particularly interested in a broad specification style called *substructural operational semantics*, or SSOS [Pfe04, PS09].[2] SSOS is a synthesis of structural operational semantics, abstract machines, and logical specifications.

One of our running examples will be a call-by-value operational semantics for the untyped lambda calculus, defined by the BNF grammar:

$$e ::= x \mid \lambda x.e \mid e_1\, e_2$$

Taking some liberties with our representation of terms,[3] we can describe call-by-value evaluation for this language with the same rewriting rules we used to describe the PDA and the Boolean program's semantics. Our specification uses three atomic propositions: one, $\mathsf{eval}(e)$, carries an unevaluated expression e, and another, $\mathsf{retn}(v)$, carries an evaluated value v. The third atomic proposition, $\mathsf{cont}(f)$, contains a *continuation frame* f that represents some partially evaluated value: $f = \Box\, e_2$ contains an expression e_2 waiting on the evaluation of e_1 to a value, and $f = (\lambda x.e)\, \Box$ contains an function $\lambda x.e$ waiting on the evaluation of e_2 to a value. These frames are arranged in a stack that grows off to the right (like the Boolean program's stack).

The evaluation of a function is simple, as a function is already a fully evaluated value, so we replace $\mathsf{eval}(\lambda x.e)$ in-place with $\mathsf{retn}(\lambda x.e)$:

$$\mathsf{ev/lam}: \quad \mathsf{eval}\,(\lambda x.e) \rightarrowtail \{\mathsf{retn}\,(\lambda x.e)\}$$

The evaluation of an application $e_1\, e_2$, on the other hand, requires us to push a new element onto the stack. We evaluate $e_1\, e_2$ by evaluating e_1 and leaving behind a frame $\Box\, e_2$ that suspends the argument e_2 while e_1 is being evaluated to a value.

$$\mathsf{ev/app}: \quad \mathsf{eval}\,(e_1\, e_2) \rightarrowtail \{\mathsf{eval}\,(e_1) \bullet \mathsf{cont}\,(\Box\, e_2)\}$$

When a function is returned to a waiting $\Box\, e_2$ frame, we switch to evaluating the function argument while storing the returned function in a frame $(\lambda x.e)\, \Box$.

$$\mathsf{ev/app1}: \quad \mathsf{retn}\,(\lambda x.e) \bullet \mathsf{cont}\,(\Box\, e_2) \rightarrowtail \{\mathsf{eval}\,(e_2) \bullet \mathsf{cont}\,((\lambda x.e)\, \Box)\}$$

Finally, when an evaluated function argument is returned to the waiting $(\lambda x.e)\, \Box$ frame, we substitute the value into the body of the function and evaluate the result.

$$\mathsf{ev/app2}: \quad \mathsf{retn}\,(v_2) \bullet \mathsf{cont}\,((\lambda x.e)\, \Box) \rightarrowtail \{\mathsf{eval}\,([v_2/x]e)\}$$

$$
\begin{array}{rrrcl}
\mathsf{eval}\ ((\lambda x.x)\ ((\lambda y.y)\ (\lambda z.e))) & & & \rightsquigarrow & \text{(by rule ev/app)} \\
\mathsf{eval}\ (\lambda x.x) & \mathsf{cont}\ (\square\ ((\lambda y.y)\ (\lambda z.e))) & & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{retn}\ (\lambda x.x) & \mathsf{cont}\ (\square\ ((\lambda y.y)\ (\lambda z.e))) & & \rightsquigarrow & \text{(by rule ev/app1)} \\
\mathsf{eval}\ ((\lambda y.y)\ (\lambda z.e)) & & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/app)} \\
\mathsf{eval}\ (\lambda y.y) & \mathsf{cont}\ (\square\ (\lambda z.e)) & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{retn}\ (\lambda y.y) & \mathsf{cont}\ (\square\ (\lambda z.e)) & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/app1)} \\
\mathsf{eval}\ (\lambda z.e) & \mathsf{cont}\ ((\lambda y.y)\ \square) & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{retn}\ (\lambda z.e) & \mathsf{cont}\ ((\lambda y.y)\ \square) & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/app2)} \\
& \mathsf{eval}\ (\lambda z.e) & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/lam)} \\
& \mathsf{retn}\ (\lambda z.e) & \mathsf{cont}\ ((\lambda x.x)\ \square) & \rightsquigarrow & \text{(by rule ev/app2)} \\
& & \mathsf{eval}\ (\lambda z.e) & \rightsquigarrow & \text{(by rule ev/lam)} \\
& & \mathsf{retn}\ (\lambda z.e) & \not\rightsquigarrow &
\end{array}
$$

Figure 1.3: SSOS evaluation of an expression to a value

These four rules constitute an SSOS specification of call-by-value evaluation; an example of evaluating the expression $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.e))$ to a value under this specification is given in Figure 1.3. Again, each intermediate state is represented by a process state or ordered context.

The SLS framework admits many styles of specification. The SSOS specification above resides in the *concurrent* fragment of SLS. (This rewriting-like fragment is called concurrent because rewriting specifications are naturally concurrent – we can just as easily seed the process state with two propositions $\mathsf{eval}(e)$ and $\mathsf{eval}(e')$ that will evaluate to values concurrently and independently, side-by-side in the process state.) Specifications in the concurrent fragment of SLS can take many different forms, a point that we will discuss further in Chapter 5.

On the other end of the spectrum, the *deductive* fragment of SLS supports the specification of inductive definitions by the same methodology used to represent inductive definitions in LF [HHP93]. We can therefore use the deductive fragment of SLS to specify a big-step operational semantics for call-by-value evaluation by inductively defining the judgment $e \Downarrow v$, which expresses that the expression e evaluates to the value v. On paper, this big-step operational semantics is expressed with two inference rules:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e_2 \Downarrow v}{e_1\ e_2 \Downarrow v}$$

Big-step operational semantics specifications are compact and elegant, but they are not particularly *modular*. As a (contrived but illustrative) example, consider the addition of a incrementing

[2]The term *substructural operational semantics* merges structural operational semantics [Plo04], which we seek to generalize, and substructural logic, which forms the basis of our specification framework.

[3]In particular, we are leaving the first-order quantifiers implicit in this section and using an informal *object language* representation of syntax. The actual representation of syntax uses LF terms that adequately encode this object language, as discussed in Section 4.1.4.

$$
\begin{array}{rrrrcl}
 & \mathsf{store}\,\underline{5} & \mathsf{eval}\,(((\lambda x.\lambda y.y)\,\mathsf{count})\,\mathsf{count}) & & \rightsquigarrow & \text{(by rule ev/app)} \\
 & \mathsf{store}\,\underline{5} & \mathsf{eval}\,((\lambda x.\lambda y.y)\,\mathsf{count}) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app)} \\
\mathsf{store}\,\underline{5} & \mathsf{eval}\,(\lambda x.\lambda y.y) & \mathsf{cont}\,(\square\,\mathsf{count}) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/lam)} \\
\mathsf{store}\,\underline{5} & \mathsf{retn}\,(\lambda x.\lambda y.y) & \mathsf{cont}\,(\square\,\mathsf{count}) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app1)} \\
\mathsf{store}\,\underline{5} & \mathsf{eval}\,(\mathsf{count}) & \mathsf{cont}\,((\lambda x.\lambda y.y)\,\square) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/count)} \\
\mathsf{store}\,\underline{6} & \mathsf{retn}\,(\underline{5}) & \mathsf{cont}\,((\lambda x.\lambda y.y)\,\square) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app2)} \\
 & \mathsf{store}\,\underline{6} & \mathsf{eval}\,(\lambda y.y) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/lam)} \\
 & \mathsf{store}\,\underline{6} & \mathsf{retn}\,(\lambda y.y) & \mathsf{cont}\,(\square\,\mathsf{count}) & \rightsquigarrow & \text{(by rule ev/app2)} \\
 & \mathsf{store}\,\underline{6} & \mathsf{eval}\,(\mathsf{count}) & \mathsf{cont}\,((\lambda y.y)\,\square) & \rightsquigarrow & \text{(by rule ev/count)} \\
 & \mathsf{store}\,\underline{7} & \mathsf{retn}\,(\underline{6}) & \mathsf{cont}\,((\lambda y.y)\,\square) & \rightsquigarrow & \text{(by rule ev/app2)} \\
 & & & \mathsf{store}\,\underline{7} \quad \mathsf{eval}\,(\underline{6}) & \rightsquigarrow & \text{(by rule ev/lam)} \\
 & & & \mathsf{store}\,\underline{7} \quad \mathsf{retn}\,(\underline{6}) & \not\rightsquigarrow &
\end{array}
$$

Figure 1.4: Evaluation with an imperative counter

counter count to the language of expressions e. The counter is a piece of runtime state, and every time count is evaluated, our runtime must return the value of the counter and then increments the counter.[4] To extend the big-step operational semantics with this new feature, we have to revise all the existing rules so that they mention the running counter:

$$\frac{}{(\mathsf{count}, \underline{n}) \Downarrow (\underline{n}, \underline{n+1})} \qquad \frac{}{(\lambda x.e, \underline{n}) \Downarrow (\lambda x.e, \underline{n})}$$

$$\frac{(e_1, \underline{n}) \Downarrow (\lambda x.e, \underline{n_1}) \quad (e_2, \underline{n_1}) \Downarrow (v_2, \underline{n_2}) \quad ([v_2/x]e_2, \underline{n_2}) \Downarrow (v, \underline{n'})}{(e_1\, e_2, \underline{n}) \Downarrow (v, \underline{n'})}$$

The simple elegance of our big-step operational semantics has been tarnished by the need to deal with state, and each new stateful feature requires a similar revision. In contrast, our SSOS specification can tolerate the addition of a counter without revision to the existing rules; we just store the counter's value in an atomic proposition $\mathsf{store}(\underline{n})$ to the left of the $\mathsf{eval}(e)$ or $\mathsf{retn}(v)$ proposition in the ordered context. Because the rules ev/lam, ev/app, ev/app1, and ev/app2 are local, they will ignore this extra proposition, which only needs to be accessed by the rule ev/count.

$$\mathsf{ev/count}: \quad \mathsf{store}\,\underline{n} \bullet \mathsf{eval}\,\mathsf{count} \rightarrowtail \{\mathsf{store}\,(\underline{n+1}) \bullet \mathsf{retn}\,\underline{n}\}$$

In Figure 1.4, we give an example of evaluating $(((\lambda x.\lambda y.y)\,\mathsf{count})\,\mathsf{count})$ to a value with a starting counter value of $\underline{5}$. This specific solution – adding a counter proposition to the left of the eval or retn – is rather contrived. We want, in general, to be able to add arbitrary state, and this technique only allows us to add *one* piece of runtime state easily: if we wanted to

[4] To keep the language small, we can represent numerals $\underline{n}$ as Church numerals: $\underline{0} = (\lambda f.\lambda x.x)$, $\underline{1} = (\lambda f.\lambda x.fx)$, $\underline{2} = (\lambda f.\lambda x.f(fx))$, and so on. Then, $\underline{n+1} = \lambda f.\lambda x.fe$ if $\underline{n} = \lambda f.\lambda x.e$.

introduce a *second* counter, where would it go? Nevertheless, the example does foreshadow how, in Part II of this dissertation, we will show that SSOS specifications in SLS allow for the modular specification of many programming language features.

An overarching theme of Part II is that we can have our cake and eat it too by deploying the *logical correspondence*, an idea that was developed jointly with Ian Zerny and that is explained in Chapter 5. In Chapter 6, we show how we can use the logical correspondence to directly connect the big-step semantics and SSOS specifications above; in fact, we can automatically and mechanically derive the latter from the former. As our example above showed, big-step operational semantics do not support combining the specification of pure features (like call-by-value evaluation) with the specification of a stateful feature (like the counter) – or, at least, doing so requires more than concatenating the specifications. Using the automatic transformations described in Chapter 6, we can specify pure features (like call-by-value evaluation) as a simpler big-step semantics specification, and then we can compose that specification with an SSOS specification of stateful features (like the counter) by mechanically transforming the big-step semantics part of the specification into SSOS. In SSOS, the extension is modular: the call-by-value specification can be extended by just adding new rules for the counter. Further transformations, developed in joint work with Pfenning [SP11], create new opportunities for modular extension; this is the topic of Chapter 7.

Appendix B puts the logical correspondence to work by demonstrating that we can create a single coherent language specification by composing four different styles of specification. Pure features are given a natural semantics, whereas stateful, concurrent, and control features are specified at the most "high-level" SSOS specification style that is appropriate. The automatic transformations that are the focus of Part II then transform the specifications into a single coherent specification.

Transformations on SLS specifications also allow us to derive abstract analyses (such as control flow and alias analysis) directly from SSOS specifications. This methodology for program abstraction, *linear logical approximation*, is the focus of Chapter 8.

1.3 Invariants in substructural logic

A prominent theme in work on model checking and rewriting logic is expressing invariants in terms of temporal logics like LTL and verifying these properties with exhaustive state-space exploration [CDE+11, Chapter 10]. In Part III of this dissertation we offer an approach to invariants that is complementary to this model checking approach. From a programming languages perspective, invariants are often associated with *types*. Type invariants are well-formedness criteria on programs that are weak enough to be preserved by state transitions (a property called *preservation*) but strong enough to allow us to express the properties we expect to hold of all well-formed program states. In systems free of deadlock, a common property we want to hold is *progress* – a well-typed state is either final or it can evolve to some other state with a state transition. (Even in systems where deadlock is a possibility, progress can be handled by stipulating that a deadlocked state is final.) Progress and preservation together imply the safety property that a language is free of unspecified behavior.

Chapter 9 discusses the use of *generative signatures* to describe well-formedness invariants

$$
\begin{array}{rrrcl}
 & & \mathsf{gen\_state} & \rightsquigarrow & \text{(by rule gen/app2)} \\
 & \mathsf{gen\_state} & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule gen/app1)} \\
\mathsf{gen\_state} & \mathsf{cont}\,(\square\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\square) & \rightsquigarrow & \text{(by rule gen/retn)} \\
\mathsf{retn}\,(\lambda y.y) & \mathsf{cont}\,(\square\,(\lambda z.e)) & \mathsf{cont}\,((\lambda x.x)\,\square) & \not\rightsquigarrow &
\end{array}
$$

Figure 1.5: Proving well-formedness of one of the states from Figure 1.3

of specifications. Generative signatures look like a generalization of context-free grammars, and they allow us to characterize contexts by a describing rewriting rules that generate legal or well-formed process states in the same way that context-free grammars characterize grammatical strings by describing rules that generate all grammatical strings.

In our example SSOS specification, a process state that consists of only a single $\mathsf{retn}(v)$ proposition is final, and a well-formed state is any state that consists of an atomic proposition $\mathsf{eval}(e)$ (where e is a closed expression) or $\mathsf{retn}(\lambda x.e)$ (where $\lambda x.e$ is a closed expression) to the left of a series of continuation frames $\mathsf{cont}(\square\, e)$ or $\mathsf{cont}((\lambda x.e)\,\square)$. We can characterize all such states as being generated from an initial atomic proposition gen_state under the following generative signature:

$$
\begin{array}{rl}
\mathsf{gen/eval}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{eval}(e)\} \\
\mathsf{gen/retn}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{retn}(\lambda x.e)\} \\
\mathsf{gen/app1}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{gen\_state} \bullet \mathsf{cont}(\square\, e_2)\} \\
\mathsf{gen/app2}: & \mathsf{gen\_state} \rightarrowtail \{\mathsf{gen\_state} \bullet \mathsf{cont}((\lambda x.e)\,\square)\}
\end{array}
$$

The derivation of one of the intermediate process states from Figure 1.3 is shown in Figure 1.5.

Well-formedness is a global property of specifications. Therefore, if we add state to the specification, we have to change the description of what counts as a final state and extend the grammar of well-formed process states. In the case of our counter extension, final states have a single $\mathsf{store}(\underline{n})$ proposition to the left of a single $\mathsf{retn}(v)$ proposition, and well-formed states are generated from an initial atomic proposition gen under the following extension to the previous generative signature:

$$
\begin{array}{rl}
\mathsf{gen/all}: & \mathsf{gen} \rightarrowtail \{\mathsf{gen\_store} \bullet \mathsf{gen\_state}\} \\
\mathsf{gen/store}: & \mathsf{gen\_store} \rightarrowtail \{\mathsf{store}(\underline{n})\}
\end{array}
$$

The grammar above describes a very coarse invariant of our SSOS specification, and it is possible to prove that specifications preserve more expressive invariants. An important class of examples are invariants about the types of expressions and process states, which will be considered in Chapter 9. For almost any SSOS specification more complicated than the one given above, type invariants are necessary for proving the progress theorem and concluding that the specification is safe – that is, free from undefined behavior. Chapter 10 will consider the use of generative invariants for proving safety properties of specifications.

1.4 Contributions

The three parts of this dissertation support three different aspects of our central thesis, which we can state as refined thesis statements that support the central thesis. We will presently discuss these supporting thesis statements along with the major contributions associated with each of the refinements.

> **Thesis (Part I):** *The methodology of structural focalization facilitates the derivation of logical frameworks as fragments of focused logics.*

The first major contribution of Part I of the dissertation is the development of *structural focalization* and its application to linear logic (Chapter 2) and ordered linear lax logic (Chapter 3). The second major contribution is the justification of the logical framework SLS as a fragment of a focused logic, generalizing the *hereditary substitution* methodology of Watkins [WCPW02].

> **Thesis (Part II):** *A logical framework based on a rewriting interpretation of substructural logic supports many styles of programming language specification. These styles can be formally classified and connected by considering general transformations on logical specifications.*

The major contribution of Part II is the development of the *logical correspondence*, a methodology for extending, classifying, inter-deriving, and modularly extending operational semantics specifications that are encoded in SLS, with an emphasis on SSOS specifications. The transformations in Chapter 6 connect big-step operational semantics specifications and the ordered abstract machine-style SSOS semantics that we introduced in Section 1.2. The destination-adding transformation given in Chapter 7 connects these specifications with the older *destination-passing style* of SSOS specification. In both chapters the transformations we discuss add new opportunities for modular extension – that is, new opportunities to add features to the language specification without revising existing rules. The transformations in these chapters are implemented in the SLS prototype, as demonstrated by the development in Appendix B.

> **Thesis (Part III):** *The SLS specification of the operational semantics of a programming language is a suitable basis for formal reasoning about properties of the specified language.*

We discuss two techniques for formal reasoning about the properties of SSOS specifications in SLS. In Chapter 8 we discuss the logical approximation methodology and show that it can be used to take SSOS specifications and derive known control flow and alias analyses that are correct by construction. The use of generative signatures to describe invariants is discussed in Chapter 9, and the use of these invariants to prove safety properties of programming languages is discussed in Chapter 10.

Bibliography

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992. 1.1, 2.1, 2.3, 2.3.1, 2.5

[BE06] Ahmed Bouajjani and Javier Esparza. Rewriting models of boolean programs. In F. Pfenning, editor, *Proceedings of the 17th International Conference on Term Rewriting and Applications (RTA'06)*, pages 136–150. Springer LNCS 4098, 2006. 1.1

[CDE+11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6)*. University of Illinois, Urbana-Champaign, 2011. 1.3, 4.6.2

[Chi95] Jawahar Lal Chirimar. *Proof Theoretic Approach To Specification Languages*. PhD thesis, University of Pennsylvania, 1995. 1.1, 2.1

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-2002-002, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003. 1.1, 2.1, 4.4, 5, 5.1, 7, 7.2, 7.2.2, B.5

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic (full-version). *Information and Computation*, 207(10):1044–1077, 2009. 1, 1.1, 2.1, 3, 4.6.2, 4.7.2

[ER12] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012. 1

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. 1.2, 2.2, 4.1, 6.3

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958. 1.1, 3, 3.1

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Principles of Programming Languages (POPL'07)*, pages 173–184. ACM, 2007. 1

[Coq10] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2010. Version 8.3. 1

[Mil93] Dale Miller. The π-calculus as a theory in linear logic: preliminary results. In

Extensions of Logic Programming (ELP'92), pages 242–264. Springer LNCS 660, 1993. 1.1, A

[Mil96] Dale Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996. 1.1

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007. 1

[Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In *Programming Languages and Systems*, page 196. Springer LNCS 3302, 2004. Abstract of invited talk. 1.2, 2.1, 5, 5.1, 6.2.2, 7.2

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Reprinted with corrections from Aarhus University technical report DAIMI FN-19. 2, 5.1, 6, 6.6.2

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer LNAI 1632, 1999. 1, 5.2

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS'09)*, pages 101–110, Los Angeles, California, 2009. 1.2, 2.1, 2.5, 2.5.2, 2.5.3, 3.6.1, 4.7.3, 5, 5.1, 6.5, 6.5.3, 7.2, 7.2.2, 10.2

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. 1

[SP11] Robert J. Simmons and Frank Pfenning. Logical approximation for program analysis. *Higher-Order and Symbolic Computation*, 24(1–2):41–80, 2011. 1.2, 2.5.3, 5.1, 6.5.3, 7, 7, 7.1, 1, 7.1, 7.2, 8, 8.3, 8.3, 8.4.2

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-2002-101, Department of Computer Science, Carnegie Mellon University, March 2002. Revised May 2003. 1.1, 1.1, 1.4, 2.2, 3.1, 3.3.3, 3.4.2, 3.8, 4, 4.1, 4.3