# Chapter 5

# On logical correspondence

In Part I, we defined SLS, a logical framework of substructural logical specifications. For the purposes of this dissertation, we are primarily interested in using SLS as a framework for specifying the operational semantics of programming languages, especially stateful and concurrent programming languages. This is not a new idea: one of the original case studies on CLF specification described the semantics of Concurrent ML [CPWW02] in a specification style termed *substructural operational semantics*, or SSOS, by Pfenning [Pfe04].

The design space of substructural operational semantics is extremely rich, and many styles of SSOS specification have been proposed previously. It is therefore helpful to have design principles that allow us to both *classify* different styles of presentation and *predict* what style(s) we should adopt based on what our goals are. In this chapter, we sketch out a classification scheme for substructural operational semantics based on three major specification styles:

- The *natural semantics*, or big-step operational semantics, is an existing and well-known specification style (and not a substructural operational semantics). It is convenient for the specification of pure programming languages.
- The *ordered abstract machine semantics* is a generalization of abstract machine semantics that can be naturally specified in SLS; this specification style naturally handles stateful and parallel programming language features [PS09].
- The *destination-passing semantics* is the style of substructural operational semantics first explored in CLF by Cervesato et al. [CPWW02]. It allows for the natural specification of features that incorporate communication and non-local transfer of control.

Each of these three styles is, in a formal sense, more expressive than the last: there are automatic and provably-correct transformations from the less expressive styles (natural semantics and ordered abstract machines) to the more expressive styles (ordered abstract machines and destination-passing, respectively). Our investigation of provably-correct transformations on SLS specifications therefore justifies our classification scheme for SSOS specifications. We call this idea the *logical correspondence*, and it is the focus of this refinement of our central thesis:

> **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.*

In this introductory chapter, we will outline our use of logical correspondence and connect it to previous work. The development of the logical correspondence as presented in this chapter, as well as the operationalization and defunctionalization transformations presented in the next chapter, represent joint work with Ian Zerny.

## 5.1 Logical correspondence

As stated above, we will primarily discuss and connect three different styles that are used for specifying the semantics of programming languages. The two styles of SSOS semantics, ordered abstract machines and destination-passing semantics, are considered because they do a good job of subsuming existing work on substructural operational semantics, a point we will return to at the end of this section. We consider natural semantics, a high-level, declarative style of specification that was inspired by Plotkin's structural operational semantics (SOS) [Plo04, Kah87], because natural semantics specifications are the easiest style to connect to substructural operational semantics. While we hope to extend the logical correspondence to other specification styles, such extensions are outside the scope of this dissertation.

While Kahn et al. defined the term broadly, natural semantics has been consistently connected with the big-step operational semantics style discussed in the introduction, where the judgment $e \Downarrow v$ expresses that the the expression $e$ evaluates to the value $v$:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ \mathsf{ev/lam} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\, e_2 \Downarrow v}\ \mathsf{ev/app}$$

Early work on natural semantics emphasized a dual interpretation of specifications. The primary interpretation of natural semantics specifications was *operational*. Natural semantics were implemented in the (non-logical) specification framework TYPOL that compiled natural semantics specifications into Prolog programs; the backward-chaining Prolog interpreter then gave an operational semantics to the specification [CDD+85]. It is also possible to view natural semantics specifications as inductive definitions; this interpretation allows proofs about terminating evaluations to be performed by induction over the structure of a natural semantics derivation [CDDK86].

The operational interpretation of natural semantics assigns a more specific meaning to expressions than the inductive definition does. For example, the rule $\mathsf{ev/app}$ as an inductive definition does not specify whether $e_1$ or $e_2$ should be evaluated in some particular order or in parallel; the TYPOL-to-Prolog compiler could have reasonably made several choices in such a situation. More fundamentally, the logic programming interpretation inserts semantic information into a natural semantics specification that is not present when we view the specification as an inductive definition (though it might be just as accurate to say that the logic programming interpretation preserves meaning that is lost when the specification is viewed as an inductive definition). The interpretation of the rules above as an inductive definition does not allow us to distinguish nontermination (searching forever for a $v$ such that $e \Downarrow v$) from failure (concluding finitely that there is no $v$ such that $e \Downarrow v$). The logic programming interpreter, on the other hand, will either succeed, run forever, or give up, thereby distinguishing two cases that are indistinguishable when the specification is interpreted as an inductive definition.

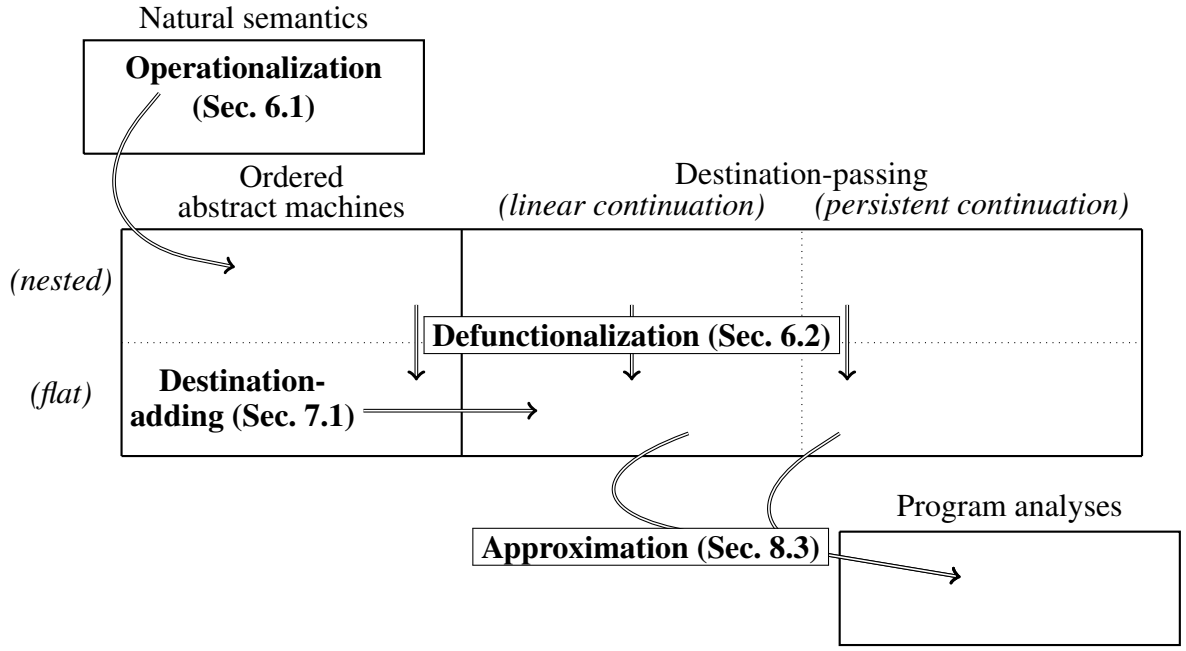


Figure 5.1: Major transformations on SLS specifications

We will present a transformation called *operationalization* from SLS-encoded natural semantics specifications into ordered abstract machines. The transformation from natural semantics to ordered abstract machines is only an instance of a much more general picture. The basic idea of operationalization is to model backward-chaining logic programming (in the sense of Section 4.6.1) as forward-chaining logic programming (in the sense of Section 4.6.2). The transformation reifies and exposes the internal structure of backward-chaining search, making evaluation order and parallelism explicit. That exposed structure enables us to reason about the difference between non-termination and failure. In turn, ordered abstract machine specifications can be transformed into destination-passing specifications by a transformation called *destination-adding*, which reifies and exposes control flow information that is implicit in the ordered context of an ordered abstract machine. Destination-passing specifications can then be transformed into a collecting semantics by *approximation*, which lets us obtain program analyses like control flow analysis. The operationalization and destination-adding transformations have been implemented within the SLS prototype. Approximation, on the other hand, requires significant input from the user and so is less reasonable to implement as an automatic transformation.

These major transformations are presented graphically in Figure 5.1 in terms of the three classification styles – natural semantics, ordered abstract machines, and destination-passing – discussed above. There are many other smaller design decisions that can be made in the creation of a substructural operational semantics, two of which are represented in this figure. One distinction, destination-passing with linear continuations versus persistent continuations, has to do with whether it is possible to return to a previous point in a program's execution and is discussed, along with first-class continuations, in Section 7.2.4.

Another distinction is between *nested* and *flat* specifications. This distinction applies to all

$$x_1{:}\langle \mathsf{p}_2(\mathsf{c})\rangle\ ord,\ x_2{:}\langle \mathsf{p}_1(\mathsf{c})\rangle\ ord,\ x_3{:}(\forall x.\, \mathsf{p}_1(x) \rightarrowtail \{\mathsf{p}_2(x) \rightarrowtail \{\mathsf{p}_3(x)\}\})\ ord,\ x_4{:}(\mathsf{p}_3(\mathsf{c}) \rightarrowtail \{\mathsf{p}_4\})\ ord$$
$$\rightsquigarrow\ x_1{:}\langle \mathsf{p}_2(\mathsf{c})\rangle\ ord,\ x_5{:}(\mathsf{p}_2(\mathsf{c}) \rightarrowtail \{\mathsf{p}_3(\mathsf{c})\})\ ord,\ x_4{:}(\mathsf{p}_3(\mathsf{c}) \rightarrowtail \{\mathsf{p}_4\})\ ord$$
$$\rightsquigarrow\ x_6{:}\langle \mathsf{p}_3(\mathsf{c})\rangle\ ord,\ x_4{:}(\mathsf{p}_3(\mathsf{c}) \rightarrowtail \{\mathsf{p}_4\})\ ord$$
$$\rightsquigarrow\ x_7{:}\langle \mathsf{p}_4\rangle\ ord$$

Figure 5.2: Evolution of a nested SLS process state

concurrent SLS specifications, not just those that specify substructural operational semantics. Flat specifications include rewriting rules $(p_1 \bullet \ldots \bullet p_n \rightarrowtail \{q_1 \bullet \ldots \bullet q_m\})$ where the head of the rule $\{q_1 \bullet \ldots \bullet q_m\}$ contains only atomic propositions. Nested SLS specifications, on the other hand, may contain *rules* in the conclusions of rules; when the rule fires, the resulting process state contains the rule. A rule $A^+ \rightarrowtail \{B^+\}$ in the context can only fire if a piece of the context matching $A^+$ appears to its left, so $(x{:}\langle \mathsf{p}_1(\mathsf{c})\rangle,\ y{:}(\mathsf{p}_1(\mathsf{c}) \rightarrowtail \{\mathsf{p}_2(\mathsf{c})\})\ ord) \rightsquigarrow (z{:}\langle \mathsf{p}_2(\mathsf{c})\rangle)$, whereas $(y{:}(\mathsf{p}_1(\mathsf{c}) \rightarrowtail \{\mathsf{p}_2(\mathsf{c})\})\ ord,\ x{:}\langle \mathsf{p}_1(\mathsf{c})\rangle) \not\rightsquigarrow$. Another example of the evolution of a process state with nested rules is given in Figure 5.2. (Appendix A gives a summary of the notation used for process states.) The choice of nested versus flat specification does not impact expressiveness, but it does influence our ability to read specifications (opinions differ as to which style is clearer), as well as our ability to reason about specifications. The methodology of describing the invariants of substructural logical specifications with *generative signatures*, which we introduced in Section 4.4 and which we will consider further in Chapter 9, seems better-adapted to describing the invariants of flat specifications.

Other distinctions between SSOS specifications can be understood in terms of nondeterministic choices that can be made by the various transformations we consider. For example, the operationalization transformation can produce ordered abstract machines that evaluate subcomputations in parallel or in sequence. In general, one source specification (a natural semantics or an ordered abstract machine specification) can give rise to several different target specifications (ordered abstract machine specifications or destination-passing specifications). The correctness of the transformation then acts as a simple proof of the equivalence of the several target specifications. (The prototype implementations of these transformations only do one thing, but the nondeterministic transformations we prove correct would justify giving the user a set of additional controls – for instance, the user could make the operationalization transformation be tail-call-optimizing or not and parallelism-enabling or not.)

The nondeterministic choices that transformations can make give us a rigorous vocabulary for describing choices that otherwise seem unmotivated. An example of this can be found in the paper that introduced the destination-adding and approximation transformations [SP11]. In that article, we had to motivate an ad hoc change to the usual abstract machine semantics. In this dissertation, by the time we encounter a similar specification in Chapter 8, we will be able to see that this change corresponds to omitting tail-recursion optimization in the process of operationalization.

Our taxonomy does a good job of capturing the scope of existing work on SSOS specifications. Figure 5.3 shows previous published work on SSOS specifications mapped onto a version of the diagram from Figure 5.1. With the possible exception of certain aspects of the SSOS pre-

sentation in Pfenning's course notes [Pfe12], the taxonomy described above captures the scope of previous work.

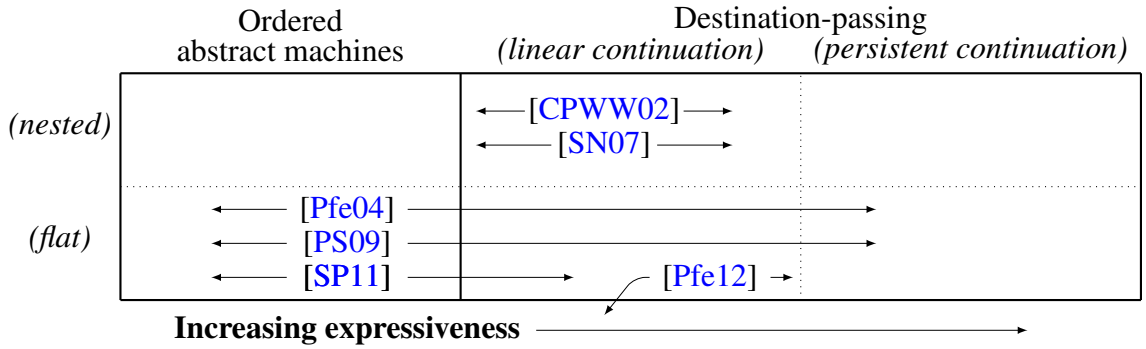


Figure 5.3: Classification of existing work on SSOS specifications

## 5.2 Related work

This part of the dissertation draws from many different sources of inspiration. In this section, we survey this related work and, where applicable, outline how our use of logical correspondence differs from existing work.

### Partiality in deductive computation

The genesis of the operationalization transformation discussed in Chapter 6 can be found in the treatment of the operational semantics of LF in Tom Murphy VII's dissertation [Mur08]; this treatment can be seen as a synthesis of the operational interpretation of natural semantics explored in Clément's et al.'s early work on natural semantics in TYPOL and the approach to theorem proving pioneered by Twelf [PS99].

In his dissertation, Murphy described a natural semantics for Lambda 5, a distributed programming language, and encoded that specification in Twelf. He then wanted to interpret that natural semantics as an *operational* semantics for Lambda 5 in the style of Clément et al., which is a natural application of Twelf's logic program interpretation [MP92]. However, Murphy also wanted to prove a safety property for his language in Twelf, and the usual approach to theorem proving in Twelf involves treating specifications as inductive definitions. As discussed above, natural semantics do not distinguish non-termination (which is safe) from failure (which indicates underspecification and is therefore unsafe).

Theorem proving in Twelf involves interpreting proofs as backward chaining logic programs that do not backtrack (recall that we called this the *flat resolution* interpretation in Section 4.6.1). Murphy was able to use the checks Twelf performs on proofs to describe a special purpose partiality directive. If a logic program passed his series of checks, Murphy could conclude that well-moded, flat resolution would never fail and never backtrack, though it might diverge. This check amounted to a proof of safety (progress and preservation) for the operational interpretation

of his natural semantics via flat resolution. It seems that every other existing proof of safety[1] for a big-step operational semantics is either classical (like Leroy and Grall's approach, described below) or else depends on a separate proof of equivalence with a small-step operational semantics.

Murphy's proof only works because his formulation of Lambda 5 is *intrinsically typed*, meaning that, using the facilities provided by LF's dependent types, he enforced that only well-typed terms could possibly be evaluated. (His general proof technique should apply more generally, but it would take much more work to express the check in Twelf.) The operationalization transformation is a way to automatically derive a correct small-step semantics from the big-step semantics by making the internal structure of a backward chaining computation explicit as a specification in the concurrent fragment of SLS. Having made this structure accessible, we can explicitly represent complete, unfinished, and stuck (or failing) computations as concurrent traces and reason about these traces with a richer set of tools than the limited set Murphy successfully utilized.

## A coinductive interpretation

Murphy proved safety for a natural semantics specification by recovering the original operational interpretation of natural semantics specifications as logic programs and then using Twelf's facilities for reasoning about logic programs. Leroy and Grall, in [LG09], suggest a novel *coinductive* interpretation of natural semantics specifications. Coevaluation $e \Downarrow^{\mathsf{co}} v$ is defined as the *greatest* fixed point of the following rules:

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

Aside from the co annotation and the different interpretation, these rules are syntactically identical to the natural semantics above that were implicitly given an inductive interpretation.

Directly reinterpreting the inductive specification as a coinductive specification doesn't quite produce the right result in the end. For some diverging terms like $\omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$, we can derive $\omega \Downarrow^{\mathsf{co}} e$ for any expression $e$, including expressions that are not values and expressions with no relation to the original term. Conversely, there are diverging terms $Div$ such that $Div \Downarrow^{\mathsf{co}} e$ is not derivable for *any* $e$.[2] As a result, Leroy and Grall also give a coinductive definition of diverging terms $e \Downarrow^{\infty}$ that references the inductively-defined evaluation judgment $e \Downarrow v$:

$$\frac{e_1 \Downarrow^{\infty}}{e_1\, e_2 \Downarrow^{\infty}} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow^{\infty}}{e_1\, e_2 \Downarrow^{\infty}} \qquad \frac{e_1 \Downarrow \lambda x.\, e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow^{\infty}}{e_1\, e_2 \Downarrow^{\infty}}$$

Now diverging expressions are fully characterized as derivations for which $e \Downarrow^{\infty}$ is derivable with an infinite derivation tree. With this definition, Leroy and Grall prove a type safety property: if $e$ has type $\tau$, then either $e \Downarrow v$ or $e \Downarrow^{\infty}$. However, the disjunctive character of this theorem means that a constructive proof of type safety would be required to take a typing derivation $e : \tau$ as

[1] Progress in particular is the theorem of concern: proving preservation for a big-step operational semantics is straightforward.

[2] Leroy and Grall discuss a counterexample due to Filinski: $Div = YFx$, where $Y$ is the fixed-point combinator $\lambda f.\, (\lambda x.\, f\, (\lambda v.\, (x\, x)\, v))\, (\lambda x.\, f\, (\lambda v.\, (x\, x)\, v))$ and $F$ is $\lambda f.\, \lambda x.\, (\lambda g.\, \lambda y.\, g\, y)\, (f\, x)$ [LG09].

input and produce as output either a proof of termination $e \Downarrow v$ or a proof of divergence $e \Downarrow^\infty$. This implies that a constructive type safety theorem would need to decide termination, and so it is unsurprising that type safety is proved classically by Leroy and Grall.

We suggest that the operationalization transformation, seen as a logical extension to Murphy's methodology, is superior to the coinductive (re)interpretation as a way of understanding the behavior of diverging evaluations in the natural semantics. Both approaches reinterpret natural semantics in an operational way, but the operationalization transformation gives us a satisfactory treatment of diverging terms without requiring the definition of an additional coinductive judgment $e \Downarrow^\infty$. And even *with* the addition of the coinductively defined judgment $e \Downarrow^\infty$, coinductive big-step operational semantics have significant issues handling nondeterministic languages, a point that we will elaborate on in Section 6.4.

## The functional correspondence

The ordered abstract machine that results from our operationalization transformation corresponds to a standard abstract machine model (a statement that is made precise Section 6.3). In this sense, the logical correspondence has a great deal in common with the *functional correspondence* of Ager, Danvy, Midtgaard, and others [ABDM03, ADM04, ADM05, Dan08, DMMZ12].

The goal of the functional correspondence is to encode various styles of semantic specifications (natural semantics, abstract machines, small-step structural operational semantics, environment semantics, etc.) as functional programs. It is then possible to show that these styles can be related by off-the-shelf and fully correct transformations on functional programs. The largest essential difference between the functional and logical correspondences, then, is that the functional correspondence acts on functional programs, whereas the logical correspondence acts on specifications encoded in a logical framework (in our case, the logical framework SLS).

The functional correspondence as given assumes that semantic specifications are adequately represented as functional programs; the equivalence of the encoding and the "on paper" semantics is an assumed prerequisite. In contrast, by basing the logical correspondence upon the SLS framework, we make it possible to reason formally and precisely about adequate representation by the methodology outlined in Section 4.4. The functional correspondence also shares some of the coinductive reinterpretation's difficulties in dealing with nondeterministic and parallel execution. The tools we can use to express the semantics are heavily influenced by the semantics of the host programming language, and so the specifics of the host language can make it dramatically more or less convenient to encode nondeterministic or parallel programming language features.

## Transformation on specifications

Two papers by Hannan and Miller [HM92] and Ager [Age04] are the most closely related to our operationalization transformation. Both papers propose operationalizing natural semantics specifications as abstract machines by provably correct and general transformations on logical specifications (in the case of Hannan and Miller) or on specifications in the special-purpose framework of L-attributed natural semantics (in the case of Ager). A major difference in this

case is that both lines of work result in *deductive* specifications of abstract machines. Our translation into the concurrent fragment of SLS has the advantage of exploiting parallelism, and also opens up specifications to the modular inclusion of stateful and concurrent features, as we will foreshadow in Section 5.3 below and discuss further in Section 6.5.

The transformation we call defunctionalization in Section 6.2, as well as its inverse, refunctionalization, makes appearances throughout the literature under various names. The transformation is not strictly analogous to Reynold's defunctionalization transformations on functional programs [Rey72], but it is based upon the same idea: we take an independently transitioning object like a function (or, in our case, a negative proposition in the process state) and turn it into data and an application function. In our case, the data is a positive atomic proposition in the process state and the application function is a rule in the signature that explains how the positive atomic proposition can participate in transitions. The role of defunctionalization within our work on the logical correspondence is very similar to the role of Reynold's defunctionalization within work on the functional correspondence [Dan08]. Defunctionalization is related to the process of representing a process calculus object in the chemical abstract machine [BB90]. It is also related to a transformation discussed by Miller in [Mil02] in which new propositions are introduced and existentially quantified locally in order to hide the internal states of processes.

The destination-adding transformation described in Section 7.1 closely follows the contours of work by Morrill, Moot, and Piazza on translating ordered logic into linear logic [Mor95, MP01]. That work is, in turn, based on van Benthem's relational models of ordered logic [vB91]. Their transformations handle a more uniform logical fragment, whereas the transformation we describe handles a specific (though useful) fragment of the much richer logic of SLS propositions.

Related work for program analysis methodology covered in Chapter 8 is discussed further in Section 8.6.

## Abstract machines in substructural logic

With the exception of our encodings of natural semantics, all our work on the logical correspondence takes place in the concurrent (rewriting-like) fragment of SLS. This is consistent with the tradition of substructural operational semantics, but there is another tradition of encoding abstract machines in substructural logical frameworks using frameworks that can be seen as *deductive* fragments of SLS. The resulting logical specifications are functionally similar to the big-step abstract machine specifications derived by Hannan and Miller, but like SSOS specifications they can take advantage of the substructural context for the purpose of modular extension (as discussed in the next section).

This line of work dates back to Cervesato and Pfenning's formalization of Mini-ML with references in Linear LF [CP02]; a mechanized preservation property for this specification was given by Reed [Ree09]. An extension to this technique, which uses Polakow's Ordered LF to represent control stacks, is presented by Felty and Momigliano and used to mechanize a preservation property [FM12]. Both these styles of deductive big-step specification are useful for creating language specifications that can be modularly extended with stateful and control features, but neither does a good job with modular specification of concurrent or parallel features.

Both of these specifications should be seen as different points in a larger story of logical

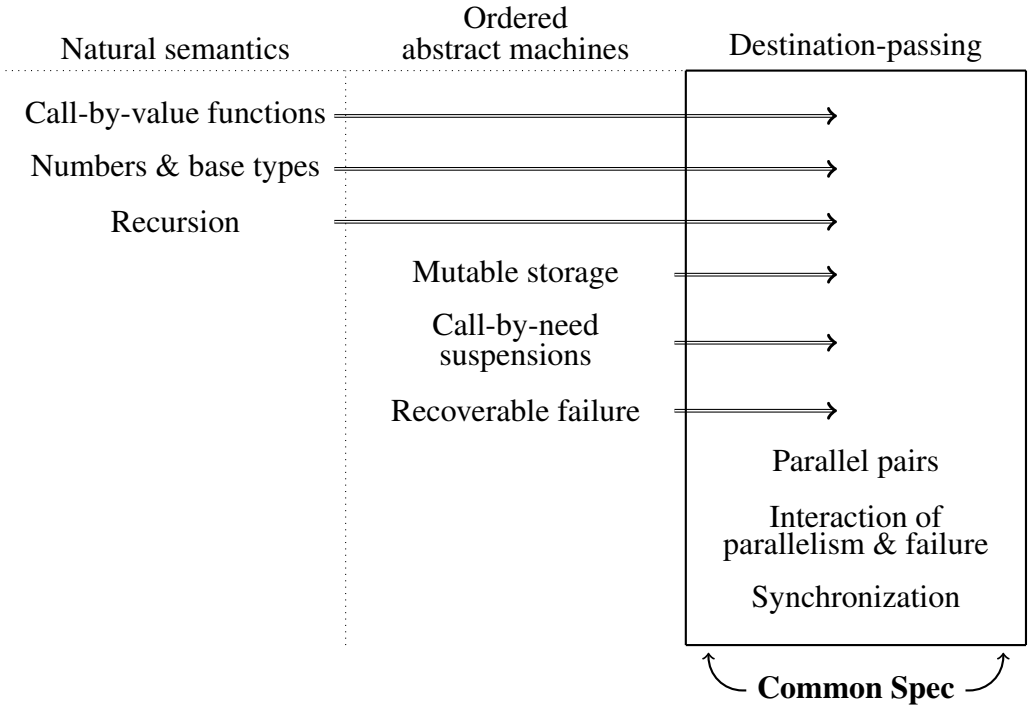


Figure 5.4: Using the logical correspondence for modular language extension

correspondence that we are only beginning to explore in this dissertation. The use of the ordered context in Felty and Momigliano's specification, in particular, is exactly analogous to the non-parallel ordered abstract machines in Chapter 6. We therefore posit the existence of a general transformation, similar to operationalization, that connects the two.

## 5.3 Transformation and modular extension

All the related work described in the previous section is concerned with *correspondence*. That is, the authors were interested in the process of transforming natural semantics into abstract machines and in the study of abstract machines that are in the image of this translation. It is possible to view the logical correspondence in the same light, but that is not how logical correspondence will be used in this document. Indeed, it is not our intent to advocate strongly for the use of natural semantics specifications at all; recall that natural semantics were used to illustrate problems with *non*-modularity in language specification in Section 1.2.

Instead, we will view the transformations illustrated as arrows in Figure 5.1 in an expressly directed fashion, operationalizing natural semantics as ordered abstract machines and transforming ordered abstract machines into destination-passing semantics without giving too much thought to the opposite direction. In the context of this dissertation, the reason that transformations are important is that they expose more of the semantics to manipulation and modular extension. The operationalization transformation in Chapter 7 exposes the order of evaluation, and the SLS framework then makes it possible to modularly extend the language with stateful features:

this is exactly what we demonstrated in Section 1.2 and will demonstrate again in Section 6.5. The destination-adding transformation exposes the control structure of programs; this makes it possible to discuss first-class continuations as well as the interaction of parallelism and failure (though not necessarily at the same time, as discussed in Section 7.2.4). The control structure exposed by the destination-adding transformation is the basis of the control flow analysis in Chapter 8.

In the next three chapters that make up Part II of this dissertation, we will present natural semantics specifications and substructural operational semantics specifications in a number of styles. We do so with the confidence that these specifications can be automatically transformed into the "lowest common denominator" of flat destination-passing specifications. Certainly, this means that we should be unconcerned about using a higher-level style such as the ordered abstract machine semantics, or even natural semantics, when that seems appropriate. If we need the richer structure of destination-passing semantics later on, the specification can be automatically transformed. Using the original, high-level specifications, the composition of different language features may appear to be a tedious and error-prone process of revision, but after transformation into the lowest-common-denominator specification style, composition can be performed by simply concatenating the specifications.

Taking this idea to its logical conclusion, Appendix B presents the hybrid operational semantics specification mapped out in Figure 5.4. Individual language features are specified at the highest-level specification style that is reasonable and then automatically transformed into a single compatible specification by the transformations implemented in the SLS prototype. In such a specification, a change to a high-level feature (turning call-by-value functions to call-by-name functions, for instance) can be made at the level of natural semantics and then propagated by transformation into the common (destination-passing style) specification.

# Bibliography

[ABDM03] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM, 2003. 5.2

[ADM04] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. 5.2

[ADM05] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. 5.2

[Age04] Mads Sig Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation (LOPSTR'04)*, pages 245–261. Springer LNCS 3573, 2004. 5.2, 6.5

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Principles of Programming Languages*, pages 81–94. ACM, 1990. 5.2

[CDD+85] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report 416, INRIA, June 1985. 5.1

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ml. In *LISP and Functional Programming (LFP'86)*, pages 13–27. ACM, 1986. 5.1

[CP02] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 179(1):19–75, 2002. 2.1, 3.3.3, 4.1, 4.1.1, 4.1.3, 4.4, 5.2, 9.1.3, 10

[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

[Dan08] Olivier Danvy. Defunctionalized interpreters for programming languages. In *International Conference on Functional Programming (ICFP'08)*, pages 131–142. ACM, 2008. Invited talk. 5.2, 5.2, 6.5.5, 6.6.2

[DMMZ12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evalu-

ation. *Theoretical Computer Science*, 435:21–42, 2012. 5.2, 6.6.1

[FM12] Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012. 5.2, 9.1.3, 10

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992. 5.2, 6.5

[Kah87] Gilles Kahn. Natural semantics. Technical Report 601, INRIA, February 1987. 5.1

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. 5.2, 2, 6.4

[Mil02] Dale Miller. Higher-order quantification and proof search. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST'02)*, pages 60–75. Springer LNCS 2422, 2002. 5.2

[Mor95] Glyn Morrill. Higher-order linear logic programming of categorial deduction. In *Proceedings of the Meeting of the European Chapter of the Association for Computational Linguistics*, pages 133–140, 1995. 5.2

[MP92] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in elf. In *Extensions of Logic Programming (ELP'91)*, pages 299–344. Springer LNCS 596, 1992. 5.2

[MP01] Richard Moot and Mario Piazza. Linguistic applications of first order intuitionistic linear logic. *Journal of Logic, Language and Information*, 10(2):211–232, 2001. 5.2

[Mur08] Tom Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2008. 5.2

[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

[Pfe12] Frank Pfenning. Lecture notes on substructural operational semantics, February 2012. Lecture notes for 15-816: Linear Logic at Carnegie Mellon University, available online: http://www.cs.cmu.edu/~fp/courses/15816-s12/Lectures/12-ssos.pdf. 5.1

[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

[Ree09] Jason Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009. 4.1, 4.1.2, 4.7.3, 5.2, 10

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*. ACM, 1972. 5.2

[SN07] Anders Schack-Nielsen. Induction on concurrent terms. In *Proceedings of the International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'07)*, pages 37–51, Bremen, Germany, 2007. 4.4, 5.1, 6.1

[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

[vB91] J. van Benthem. *Language in Action: Categories, Lambdas and Dynamic Logic*, volume 130 of *Studies in Logic and the Foundations of Mathematics*, chapter 16, pages 225–250. North-Holland, Amsterdam, 1991. 5.2