An Introduction to Operational Semantics

Sanjiva Prasad S. Arun-Kumar

February 4, 2002

Abstract

The objective of this chapter is to introduce to compiler developers the rudimentary concepts of *operational semantics* used in specifying the operational behavior of programs and systems, and for reasoning about them. There are already various excellent comprehensive introductions to syntax-directed approaches to operational semantics, most notably the seminal papers by Plotkin [Plo81] and Kahn [Kah87]. Some of that material has already been incorporated in standard text books on the semantics of programming languages and concurrency, such as those by Winskel [Win93], Gunter [Gun92], Watt [Wat90] and Hennessy [Hen88]. Yet, though the concepts and techniques employed are mathematically simple and accessible, many compiler developers have not been exposed to them.

The material presented here is largely based on the seminal work mentioned above, and is aimed at presenting the ideas in an integrated form. It is tutorial in nature, oriented towards those interested in relating language specification to compiler design. There are also several excellent surveys and references on research aspects in operational semantics, particularly in the context of semantics of computation [Ong99] and of process algebra [AFV00], intended for those who are already familiar with semantics issues in programming languages and concurrency.

Contents

1 Introduction

Operational Semantics involves giving a precise description of the behavior of a program or a system, namely, how it may execute or operate. As in any semantic enterprise, the intention in developing operational semantics is to give behavioral descriptions in rigorous mathematical terms, in a form that supports understanding and reasoning about the behavior of the systems under consideration. A mathematical model serves as the basis for analysis and verification. In fact, the very act of formalization can help remove misconceptions and focus attention on subtleties that may be glossed over in an informal description.

A clear operational semantics is an invaluable reference while developing language implementations, as was recognized over a quarter of a century ago by McCarthy [McC63], Landin [Lan64, Lan65b], Hoare and Lauer [HL74], Milner, Plotkin and various other researchers. Early examples of real-world languages being provided formal operational semantics include Algol 60 [Lau68] and PL/I [PL/86].

Formalism, *per se*, is not the only goal; defining the meaning of a programming language as the behavior induced by a particular implementation *is* a formal treatment. However, such an approach is not particularly satisfactory since the intention is to provide behavioral descriptions at a high level, divorced from implementation details to as great an extent as possible. Moreover, the high-level formalism should be readily accessible. Indeed, the attraction of using operational semantic approaches to programming languages is the relative simplicity of the formal mathematics and the associated techniques.

The past twenty or so years has seen, following seminal contributions by Plotkin [Plo81], Milner [Mil73, Mil76], Kahn [Kah87], Hoare [Hoa85] and others, the development of syntax-directed "structural" frameworks that provide, to quote Plotkin, a "simple and direct method for specifying the semantics of programming languages", which require very little mathematical background, that yet provide "concise, comprehensible semantic definitions". The definition of the mostly functional language Standard ML in a wholly operational semantic framework [MTHM97] is an excellent example of the power and versatility as well as the relative accessibility of these operational techniques. Other languages which have complete operational descriptions are Esterel [BC84, Gon88, BG92] and Ada ([ANB+86] contains an early definition that employs the main ideas discussed here in a rigorous algebraic framework).

While formalization is clearly important for research in programming language semantics, the aim of this chapter is to make modern approaches to operational semantics accessible to those involved in compiler design and development. It is therefore worthwhile to reiterate here why formal operational descriptions are important in the context of compiler design and implementation. As mentioned above, such descriptions provide an unambiguous definition of a language, which can serve as a reference for implementations. A structural operational semantics (SOS) style seems to be an increasingly favored style of providing a comprehensive and comprehensible formal definitions of programming languages. Apart from the examples of

Standard ML and Esterel cited above, SOS semantics have been provided for several languages including Java [CKRW99] and logic programming languages based on Prolog [HJP92]. Secondly, these formal descriptions allow us to develop theories such as program equivalence or orderings, which serve as a semantically sound basis for assessing proposed *program optimizations* and *static analysis techniques*. While it may be naive to expect the algebraic laws of equivalence (or ordering) to suggest optimizations, it is nevertheless expected that any optimization preserves the operational behavior of a program (or at least the important behavioral properties needed in the context of a particular computation). Thirdly, the operational descriptions give us a framework in which compiler verification can be formulated and carried out [dS92]. Finally, operational frameworks allow us to explore novel, alternative implementation techniques — by studying different abstract implementations that realize the same specifications. A noteworthy approach in this respect is that of Hannan and his collaborators [Han94].

Structural operational techniques have been successfully employed with great success for studying the correctness of compiler techniques and hardware implementations [Tin01, WBB92, WO92], for compiler verification [dS92, HP92], for establishing *type soundness* following the work of Wright and Felleisen [WF94], for static program analysis [MS96] and deriving proof rules for functional languages [San97].

We also should mention that there are areas of crucial importance to compiler developers where operational techniques have not been seriously applied. An example is floating-point computation, where to our knowledge, the intricacies of the numerical models proposed and used have not been adequately addressed in an operational framework.

Operational descriptions at different levels of abstraction. Formal operational descriptions of program execution can be presented in several different ways. In fact, having different descriptions may serve a useful purpose, especially since they are usually presented at varying levels of abstraction. In the following paragraphs, we give a brief overview of three broad levels of operational description, which have historically tended (roughly) to go from "low-level" to "high-level" descriptions for the same language, though there have been notable exceptions where implementations have been guided by higher-level specifications.

The very first step in providing a description of a language independent

of any particular implementation is to concentrate on the abstract syntactic structure of programs in the language rather than on the concrete syntax. This also has the advantage of being able to abstract over different concrete renderings of a concept in different languages, *e.g.*, the syntax used for assignment in Pascal versus that used in C. A relatively higher level semantic description than a particular implementation is achieved by *translation* of the abstract syntax into instructions of a simple machine, the description of which is given in abstract terms, typically as a finite collection of rules. Such an idealized machine is called an *abstract machine*.

Reasoning about programs using abstract machine descriptions consists of reasoning about the process of translation, and then reasoning about execution sequences of the abstract machine. A significant observation that greatly simplified the first aspect was the following: The abstract syntax of most languages is inductively characterized, and the translation to the machine instructions tends to be a mapping that preserves the abstract syntactic structure, often a *homomorphic* function.

A good early example of this kind of operational description is Landin's use of the so-called *SECD* machine to specify the operational semantics of a quintessential (call-by-value) functional language ISWIM [Lan65a]. Also well known is the Warren Abstract Machine (WAM) [War83], used to specify the execution machinery for Prolog. Abstract machines are a popular (and often the first) method for specifying the execution semantics of a proposed language as well as for outlining an implementation. For instance, abstract machines were used in presenting the first formal operational descriptions of various extensions to the functional paradigm such as integrations of functional programming with concurrent programming models based on ideas from process algebra [Car86a, Car86b, GMP89].

Although abstract machines provide higher-level, implementation independent specifications of program execution, it is not always clear how effective such techniques are in proving program properties, notions of program equivalence and developing a semantically-justified algebra of programs. Moreover, proofs about program execution are (often tedious and cumbersome) induction arguments on execution sequences, using case analyses on which rule is employed at each step, with little reference to the original *source programs* and their structure.

A second and novel step was the development of *structural operational semantics*, or "SOS", where program behavior was expressed directly in terms of the source programs (and perhaps a few ancillary data structures) without

any intervening translation to an abstract machine. The structural approach consists of providing an inductive definition of a relation describing program execution, which follows the inductive structure of the abstract syntax. Thus, in the operational setting, the approach adheres to a *compositionality principle* associated with Frege that "the meaning of a phrase can be obtained from the meaning of its components in a well-defined way", a feature of the Scott–Strachey style of denotational semantics. The standard presentation of the inductively defined relation is by using inference rules. The consequent of a rule defines a transition from a compound expression, which depends on the transitions for one or more of its components specified in the rule antecedent. This inductive approach based on abstract syntactic structure is also appropriate for formulating static semantics. An added bonus of using *relations* is that features such as non-termination and partiality, non-determinism, error configurations and various others can easily be accommodated into the framework without having to resort to more difficult mathematical concepts.

The associated proof techniques are based on induction on the proof trees built using the inference rules, or equivalently — since the inference rules are presented in a syntax-directed manner — on the structure of the source program. Notions of program equivalence or ordering are stated directly in terms of the source programs rather than via any other machinery, and thus the development of an algebra of programs gets facilitated. It is this aspect of structural induction that justifies the moniker "structural", since the other techniques also ultimately depend on program structure.

The pioneering works where the structural approach is articulated are those of Plotkin [Plo81, Plo83], Milner [Mil80] and Kahn [Kah87], although instances of the structural approaches predate these publications — most notably, the operational semantics of various λ-calculi [Bar84]. Structural semantics comes in a variety of flavors, and we broadly classify them as: (i) "big-step", often called "natural" due to its connections with normalization in Natural Deduction proof systems [Kah87, BH87], and sometimes relational [MTHM97] or proof-theoretic [MH90]; and (ii) "small-step", which is often called "reduction" following the terminology used in the λ-calculus [Plo81]. Big-step semantics justify a complete execution sequence using a tree-structured proof whereas small-step semantics provide tree-structured justifications for each step of the sequence. There are, however, situations where a "mixed-step" formulation is convenient. In contrast, abstract machine semantics consists of a sequence of steps, each justified as being an instance of a *conditional rewrite rule*.

Yet another dimension in the varieties of structural operational semantics is the use of *labelled* relations that allow the specification of the interaction between a program and its environment during execution. Most examples of labelled relations are in a small-step style, and abstract machines rarely use labelled relations at all.

One of the aims of this chapter is to convey to the reader the rudiments of these three kinds of operational semantics and their inter-relationships and important syntactic properties, such as confluence and standardization. We endeavor to present these notions in frameworks that are as simple and familiar as possible, and assuming minimal concepts. Various aspects of these connections have been studied in great detail elsewhere, assuming varying degrees of familiarity with the concepts. Plotkin [Plo81] covers a large variety of constructs in the reduction semantics framework. Some subtle issues arising in relating the big-step and small-step formulations are explored in [Ast91]. Winskel's book [Win93] studies the relation between big-step and denotational semantics for simple imperative and functional languages. Hannan and Miller [MH90] present a framework for constructing abstract machines from big-step semantics for functional languages via a series of correctness preserving transformations. Hannan further explores concrete realizations of the machines [Han91]. Plotkin [Plo75] studies the connection between the reduction semantics of the call-by-value λ-calculus and its abstract machine (and respectively for call-by-name), as well as how the calculi relate to one another by continuation-passing style (CPS) translations. An excellent reference covering much of this material in detail is [AC98].

Disclaimers. This chapter does not attempt to survey the variety of operational semantics frameworks used in the specification and implementation of programming languages. In particular, two major approaches have been neglected — those of *Action Semantics* [Mos92] and *Evolving Algebras* or *Abstract State Machines* [Gur93]. Action Semantics is based on ideas from universal algebra, and seeks to combine the salient strong features of denotational and operational approaches, without their weaknesses. The semantic specification is given round the basic *actions* in a system, and the approach addresses the important issues of readability and modularity of semantics frameworks. In the sequel, we will see that even small language extensions necessitate major changes in the semantic rules. Action semantics has been successfully used in diverse applications, a very significant one in the area of

compilation being the work of Palsberg on provably correct compiler generation [Pal92].

Evolving algebras, or abstract state machines as they are called now, are based on the idea of interpreting the dynamic semantic actions of a system as operators of an algebra that evolves during execution. The approach is very general and permits specification of a system at different levels of abstraction. The operational framework is closely related to conditional rewriting systems, and the theory also addresses the mathematical issue of algebraic models for rules. Furthermore, abstract state machine descriptions admit parallelism (concurrency) in an extremely natural way. They have been used extensively for describing a variety of systems and languages, such as Prolog [BS90] and Modula-2 [Mor88], apart from being used as a vehicle for understanding various concurrency features of Ada and other such intricacies.

We have also concentrated on only three paradigms — imperative, functional and concurrent — and not addressed issues in logic programming and object-oriented programming. Nor have we examined seriously the issues that arise when different such paradigms are integrated in a single language.

Relationship with other kinds of semantics. An alternative to operational techniques for specifying the semantics of programming languages is providing mathematical models, *i.e.*, *denotational semantics* (well known text books on denotational semantics are [Sto77, Sch86]). Denotational frameworks are also specified inductively on abstract syntax. The attraction of denotational methods is that they provide rich mathematical theory for reasoning about programs. Moreover, when the denoted objects are readily constructible in a computational framework, the semantics can be viewed as providing an immediate implementation of the language.

However, two questions immediately arise when providing a language with a denotational model: First, whether such a model is in (complete) agreement with operational intuition. Milner was the first to propose a criterion, called *full abstraction*, which formalizes this notion of "complete agreement" between the two forms of semantics. He convincingly argues that it is the operational semantics that should be the reference (the "touchstone") for assessing mathematical models, rather than the converse, since operational models are (usually) set up with minimal preconceptions. The second question is whether there is indeed a unique mathematical model. Milner points out that any mathematical model can capture only some aspects of the opera-

tional behavior, whereas there may be diverse aspects that can be of interest — especially in non-deterministic computations. Operational frameworks, being relational, can easily accommodate aspects such as non-determinism, partiality, erroneous computations, etc., with minimal reworking of definitions, whereas these may necessitate significant changes to the mathematical models used in a denotational description.

Another alternative to the operational approach is the so-called *axiomatic semantics* [Hoa69] in which the meaning of a programming construct is given using *proof rules* within a *program logic*. The orientation of the approach is towards proving program correctness with respect to logical specifications. Again, one could argue for the primacy of operational techniques to interpret and justify the soundness of the logical rules. Moreover, the formulation of operational semantics using inference rules in the SOS approaches together with the induced algebraic notions of equivalence or ordering on programs incorporate many aspects of the axiomatic approach into operational ones — compositionality, syntax-orientation and proof theory in particular.

It must be noted, however, that the three approaches are not mutually exclusive or conflicting. Each finds use when reasoning about programs, and often while employing a particular kind of approach, one may resort to another. For instance, while reasoning about the operational semantics (proving meta-theorems) it may be convenient to use results from the denotational semantics since this enables one to abstract away irrelevant operational details and to use abstract mathematical concepts.

Structure of this chapter. The rest of the chapter is structured as follows. In §2, we introduce various important rudimentary concepts used in describing the operational behavior of systems. We start with the notion of transition systems, and then proceed to providing meaning to abstract syntax trees. We use a simple language of expressions to illustrate three different levels of operational description. We enrich the language with variables and then scoped local definitions. §3 presents the operational semantics for a simple imperative language. A variety of extensions of this language to incorporate non-determinism and parallel execution, block structure, simple procedures and storage allocation are discussed. In §4, we discuss descriptions of higher-order functions, referring to the λ-calculus and two evaluation strategies — call-by-name and call-by-value, together with environment machines for implementing these calculi. Then, in §5, we mention features of

languages involving concurrency and interaction that are naturally modeled using labelled transitions, before concluding in §6.

2 Preliminaries

2.1 Transition Systems

The primary task involved in providing an operational description of a system is to specify the *configurations* of the system and the possible *transitions* between configurations. A *transition system* consists of a collection (usually a set) $\mathcal{S}$ of configurations and a binary relation on configurations $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ called the transition relation. We use the metavariable s to range over configurations. In most applications, a subset $\mathcal{I} \subseteq \mathcal{S}$, called *initial* or *starting* configurations, is distinguished. *Terminal configurations* are those from which a transition is not possible — $\{s \in \mathcal{S} \mid \nexists s' : s \longrightarrow s'\}$. We denote the transitive closure of the transition relation by $\longrightarrow^+$ and its reflexive transitive closure by $\longrightarrow^*$. Termination arguments often require showing that the transition relation is well-founded.

A closely related notion is that of a *labelled transition system* (LTS). Let $\mathcal{L}$ be a set of *labels*, with l a typical label. An LTS consists of a set of configurations $\mathcal{S}$, the label set $\mathcal{L}$, and a relation $\_ \stackrel{\_}{\longrightarrow} \_ \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ called the labelled transition relation. We write $s \stackrel{l}{\longrightarrow} s'$ to mean that $\langle s,\ l,\ s' \rangle \in \_ \stackrel{\_}{\longrightarrow} \_$. Often a LTS is presented as a collection of TSs sharing the same configurations $\mathcal{S}$, but with one transition relation for each label.

Example 2.1 (Lexical Analysis) *Lexical analysis can be cast as a transition system. Let $\mathcal{M} = \langle Q, q_0 \in Q, \delta \subseteq Q \times \Sigma \times Q, F \subseteq Q \rangle$ be a finite state automaton recognizing a language over alphabet Σ, and let $\varsigma \in \Sigma^*$ be any string over that alphabet. Let ϵ denote the empty string, and let $a\varsigma$ denote the string starting with letter a followed by the suffix string ς.*

Let $\mathcal{S} = Q \times \Sigma^*$ and let $\longrightarrow$ be defined as $\langle q,\ a\,\varsigma \rangle \longrightarrow \langle q',\ \varsigma \rangle$ if and only if $(q, a, q') \in \delta$. $\mathcal{I} = \{\langle q_0,\ \varsigma \rangle \mid \varsigma \in \Sigma^*\}$ is the set of initial configurations. Terminal configurations are of two kinds: those in $F \times \{\epsilon\}$ are "accepting" whereas those in $(Q - F \times \{\epsilon\}) \bigcup \{\langle q,\ b\,\varsigma \rangle \mid \neg \exists q' : (q, b, q') \in \delta\}$ are "non-accepting".

Example 2.2 (An automaton is an LTS) *Finite State (and indeed other) Automata are examples of labelled transition systems, with the configurations*

$\mathcal{S}$ being the states, labels being Σ and δ being the labelled transition relation.

The example of automata also motivates a bunch of concepts important in operational semantics. We usually associate a notion of *observation* with a transition system (*e.g.*, consumption of a string and terminating in an accepting state in a finite state automaton), with respect to which transition systems are ascribed *observable behaviors* (*e.g.*, strings accepted by the automaton). There can be different notions of what is observable for even the same transition system. Any given notion of observability yields a corresponding notion of equivalence or ordering between two transition systems based on their observable behaviors.

Definition 2.3 (observational equivalence and ordering) *TS_1 is said to be observably simulatable by TS_2, written $TS_1 \preceq TS_2$, if every observable behavior possible of TS_1 is also possible of TS_2. TS_1 and TS_2 are considered equivalent, denoted $TS_1 \approx TS_2$, if both have the same observable behaviors.*

Equivalence of two systems does not necessarily imply that one can be replaced by the other in any context, since some notions of equivalence may not be preserved under each and every construction possible in a class of transition systems.

The automata example also give an idea of how a LTS can relate to a TS. The automaton LTS describes the control aspect of the transition system in abstraction from the data (the string ς) on which it is run. The dichotomy between control and data is not the central issue, and is indeed relevant to this and some other examples. Rather, labels are used to indicate *interaction* between a component of a larger system with its context. This interaction can be of a variety of kinds, and hence there are diverse uses of labelled transition systems. For example, a process receiving signals and performing some computation in response can be specified separately from the processes sending it signals. The use of labelled transitions permits the description of a component's behavior separately from that of its context, with the labels specifying the interaction capabilities. Very crudely, a labelled transition system can be turned into a corresponding unlabelled one by by providing within the system "enough context" — thus "closing up" a description of an open system. Conversely, contexts can be used to label transitions. The main issue is to characterize interesting decompositions of systems into program fragment and context. This is still very much the subject of active research, with some recent promising results in this direction [Lei01, Sew98].

Example 2.4 (Parsing) *We also encounter a transition system in parsing. String generation can be thought of as a transition system as follows. Let* $\mathcal{G} = \langle \mathsf{N}, \mathsf{T}, \mathsf{P}, S \in \mathsf{N} \rangle$ *be a context-free grammar. Let the configurations* $\mathcal{S} = (\mathsf{N} \bigcup \mathsf{T})^*$ *and let the transition relation* $\longrightarrow$ *be defined as* $s \longrightarrow s'$ *if and only if there exists a production* $r \in \mathsf{P}$ *such that* $r \equiv X \rightarrow w$ *for some* $X \in \mathsf{N}$ *and* $w \in (\mathsf{N} \bigcup \mathsf{T})^*$, $s = s_1 X s_2$ *and* $s' = s_1 w s_2$. *S is the unique starting configuration, and those terminal configurations that are in* T^* *and reachable from S are the "generated" strings.*

This transition system can be "reversed" to yield a transition system for parsing. The production rules are used in the reverse direction; $\mathcal{I} = \mathsf{T}^*$ *is the set of initial configurations, there is a single "accepting" final configuration S, and possibly many other terminal configurations that are "non-accepting".*

Remark 2.5 *Plotkin's seminal paper [Plo81, chapter 1] lists several different examples of transition systems or labelled transition systems that one encounters in computer science — finite state automata, transducers, grammars of different types, k-counter machines, stack machines, Petri Nets, Turing Machines, Semi-Thue systems, Post systems, L-systems, Conway's Game of Life, push down automata, tree automata, cellular automata, neural nets. In addition, many dynamic systems we encounter in daily life may be modeled as transition systems. Games are good examples of transition systems.*

Properties. Transition systems provide a framework on which we can drape various formal verification exercises. Many of these involve establishing that a particular transition system satisfies various kinds of properties. One such important property is *totality*. A transition system is *total* if it has no terminal configurations, *i.e.*, for every $s \in \mathcal{S}$ there exists $s' \in \mathcal{S}$ such that $s \longrightarrow s'$. Another common property is *determinism*: for every $s \in \mathcal{S}$, $|\{s' \mid s \longrightarrow s'\}| \leq 1$. These notions can also extend to labelled transitions, either "per-label" or "across labels".

A crucial property in the above examples for lexical and grammatical analysis is *reachability* from designated initial configurations. Reachability is also used in proving *safety properties* of systems — no "bad" configuration is reachable from specified initial configurations.

Another property is what we call *properly terminating*, where all terminal configurations are "good". This is an example of a *liveness property* — that "something good can eventually happen".

Another important meta-property is *confluence*: for any $s, s_1, s_2 \in \mathcal{S}$, whenever $s \longrightarrow^* s_1$ and $s \longrightarrow^* s_2$, then there exists $s_3 \in \mathcal{S}$ such that $s_1 \longrightarrow^* s_3$ and $s_2 \longrightarrow^* s_3$. Stronger confluence properties are the so-called "diamond" properties. A transition system exhibits the "*strong diamond*" property if for any $s, s_1, s_2 \in \mathcal{S}$, whenever $s \longrightarrow s_1$ and $s \longrightarrow s_2$ and $s_1 \neq s_2$, then there exists $s_3 \in \mathcal{S}$ such that $s_1 \longrightarrow s_3$ and $s_2 \longrightarrow s_3$. The transition system has a "*weak diamond*" property if whenever $s \longrightarrow s_1$ and $s \longrightarrow s_2$ and $s_1 \neq s_2$, then s_3 is reachable from s_1 and s_2 via the reflexive transitive closure of the transition relation, that is, $s_1 \longrightarrow^* s_3$ and $s_2 \longrightarrow^* s_3$.

Various properties follow from certain finiteness constraints on transition systems. A TS (or LTS) is called

- *finitely branching* if for every $s \in \mathcal{S}$, the set $\{s' \mid s \longrightarrow s'\}$ is finite.
- *finite* if it is finitely branching and $\longrightarrow$ is a well-ordering.
- *regular* if it is finitely branching and for each $s \in \mathcal{S}$, the set $\{s' \mid s \longrightarrow^* s'\}$ is finite, where $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$.

In general, transition systems whose transition relation can be characterized in a concise but abstract manner (usually as a set of rules) are of interest, since they usually admit effective techniques for establishing properties of those systems. *Finite* or *inductively* characterized transition systems are extremely common, with induction and case analysis on (linear) sequences of transitions being the most widely wielded proof methods for reasoning about execution sequences or, at a higher level, observable behavior.

2.2 Structural Operational Semantics for Expressions

Abstract syntax. It is the *abstract* rather than the concrete syntax of a language that is of interest while specifying the meaning of programs. Operational semantics descriptions manipulate these abstract syntactic objects and work wholly within syntax. For convenience, however, it may be necessary to augment the syntax with "extra-syntactic" data structures, but these entities can be shown to correspond in some obvious way to purely syntactic entities. The abstract syntax of programs can be inductively characterized, *e.g.*, as trees. We will use abstract grammars as a handy notational device for describing abstract syntactic categories.

We present three different kinds of operational description for an extremely simple language *Exp*; the presentation can be adapted to any language of first-order expressions.

Example 2.6 (Simple arithmetic expressions) *Let Num denote the denumerable set of numerals (in some radix), and let $\mathcal{X}$ be a denumerable set of variables, with x, y, z typical meta-variables ranging over $\mathcal{X}$. Exp can be presented using the following abstract grammar, where e, e_1, e_2 are meta-variables ranging over Exp, and n ranges over Num.*

$$e \in \mathit{Exp} \quad ::= \quad x \mid n \mid (e_1 + e_2)$$

Expression evaluation consists of simplifying a given expression to a form that cannot be further simplified, hopefully to an element in a set of "good" canonical forms that we loosely call "values" (there are a variety of notions of "value" depending on the language). The first task in presenting operational semantics for expressions is to identify the set $\mathcal{V}$ of values. In the next few examples the set $\mathcal{V}$ will be the set of numerals *Num*. The meta-variable v ranges over $\mathcal{V}$.

For expressions containing variables, we need to know what the variables stand for in order to simplify them to values. Accordingly, we present the operational semantics with respect to a finite domain function called an *environment* $\gamma : \mathcal{X} \rightarrow_{\mathit{fin}} \mathcal{V}$, that maps variables to values. Let *Env* denote the set of such finite domain functions from variables to values[1]. Environments are an example of "extra-syntactic" constructions we employ in our operational description. We write $dom(\gamma)$ to mean the set $\{x \in \mathcal{X} \mid \gamma(x) \text{ defined}\}$. We work with finite domain functions since it is inappropriate to frame essentially syntactic ideas in terms of infinite structures. If γ_1 and γ_2 are finite domain functions, we denote by $\gamma_1[\gamma_2]$ the finite domain function with domain $dom(\gamma_1) \bigcup dom(\gamma_2)$ defined as

$$\gamma_1[\gamma_2](x) = \begin{cases} \gamma_2(x) & \text{if } x \in dom(\gamma_2) \\ \gamma_1(x) & \text{if } x \in dom(\gamma_1) - dom(\gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

[1]It is also possible to work with environments which are finite domain functions from variables to variable-free expressions rather than to values. The nature of the rules and results does not change, except perhaps in some minor details.

$$(var)\quad \frac{}{\gamma \vdash x \Longrightarrow_e \gamma(x)} \qquad \text{where } x \in dom(\gamma)$$

$$(num)\quad \frac{}{\gamma \vdash n \Longrightarrow_e n}$$

$$(add)\quad \frac{\gamma \vdash e_1 \Longrightarrow_e n_1 \quad \gamma \vdash e_2 \Longrightarrow_e n_2}{\gamma \vdash (e_1 + e_2) \Longrightarrow_e n_3} \qquad \text{where } n_3 = ADD(n_1, n_2)$$

Table 1: "Big-step" semantics for evaluating simple expressions

Big-step or Natural Semantics We first present a "big-step structural operational semantics" or "natural semantics" for *Exp*.

The "big-step" transition relation $\Longrightarrow_e \subseteq Env \times Exp \times \mathcal{V}(= Num)$ is defined inductively as the smallest relation closed under the inference rules given in Table 1. We read the relation $\gamma \vdash e \Longrightarrow_e n$ as "given environment γ, expression e can evaluate to value n". When the environment γ is not needed, and so can be arbitrary, we sometimes omit writing "$\gamma \vdash$".

This relation can be viewed as a transition system with configurations $\mathcal{S} = (Env \times Exp)$. A transition $\gamma \vdash e \Longrightarrow_e v$ is understood as a transition $\langle \gamma, e \rangle \rightarrow \langle \gamma, v \rangle$, highlighting the fact that transitions leave γ unchanged.

The way these rules are used is that if we have an expression that *matches* the left side of the *consequent* ("denominator") of a rule via a substitution ρ for the schematic variables, and if using the same substitution ρ, all the *antecedents* (statements in the "numerator") can be inductively established while also respecting any side-conditions, then the expression *can* evaluate to an expression of the form given on the right side of the *consequent* instanced using ρ. Used in this manner, the rules can be seen as forming tree-structured justifications or *proof trees* of why an expression e can evaluate to a value n — the goal judgment ($e \Longrightarrow_e n$ in this case) is at the root, the leaves are axiom instances, and internal nodes correspond to rule instances with a branch for each antecedent.

The use of proof rules to specify transition systems is itself an area of research. [AFV00] contains an excellent summary of rule specifications, the meanings of the transition systems they specify and of various formats and the formal properties they guarantee (see also [Mic94]).

Observe that the rules are *syntax-directed*, in that there is a rule for each

syntactic case. Further, in rules with antecedents, the consequent of the rule describes the evaluation of a compound expression; this evaluation depends on the evaluation of the component subexpressions, described in the rule's antecedents. The base cases of the relation $\Longrightarrow_e$ are the axioms (*num*) and *var*, which state(respectively) that any numeral evaluates to itself, since it is in canonical form, and that a variable evaluates to the value associated with it in the environment. Note, however, that instances of the rule *var* apply only when the side condition or *proviso* $x \in dom(\gamma)$ holds. The induction case is the rule (*add*). The rule may be read as "given γ, expression $(e_1 + e_2)$ can evaluate to numeral n_3 if expression e_1 can evaluate to a numeral n_1 with respect to γ, and e_2 to n_2 also with respect to *gamma*, and where adding numerals n_1 and n_2 yields numeral n_3. We assume there is a syntactic routine *ADD* for adding numerals.

Note that the big-step relation is reflexive on values. The relation is not total on environments and *Exp*, because the *var* rule does not specify how to evaluate a variable $y \notin dom(\gamma)$.

Typical exercises involve studying various properties of this relation. For instance, assuming that the procedure *ADD* is functional and total, we can show that the relation $\Longrightarrow_e$ is indeed a *partial function*:

$$|\{n \mid \gamma \vdash e \Longrightarrow_e n\}| \leq 1 \text{ for all } \gamma \in Env \text{ and } e \in Exp.$$

If $vars(e)$ is the set of variables in e, we can show:

Proposition 2.7 *For any $e \in Exp$, $\gamma \in Env$, if $vars(e) \subseteq dom(\gamma)$, there exists $n \in Num$ such that $\gamma \vdash e \Longrightarrow_e n$.*

Proof of this proposition is by induction on the structure of the proof tree of $\gamma \vdash e \Longrightarrow_e n$, which amounts to induction on the structure of e, since the relation is syntax-directed.

Further, we can show that the big-step operational semantics agrees with any "standard" denotational semantics if the procedure *ADD* behaves in accordance with the corresponding mathematical operation. Let ρ be an assignment of values to variables, let $[\![n]\!]$ denote the number represented by numeral n, and let $[e]\rho$ be the denotation of e with respect to ρ.

Proposition 2.8 *For any $e \in Exp$, γ, ρ such that $vars(e) \subseteq dom(\gamma)$ and for all $x \in dom(\gamma)$, $\rho(x) = [\![\gamma(x)]\!]$: $\gamma \vdash e \Longrightarrow_e n$ if and only if $[e]\rho = [\![n]\!]$.*

This result too is proven by induction on the structure of e.

Small-step or Reduction Semantics. The big-step relation *specifies* what normal forms an expression may have. It is a high-level specification, possibly non-deterministic, and does not detail how the computation may be performed. It is inherently parallel; for example, in simplifying $(e_1 + e_2)$, no indication is given as to whether to simplify e_1 before e_2 or otherwise. Nor is any hint given on how to implement the relation with finite resources.

In contrast, a *small-step* or *reduction* relation is used to specify not merely what an evaluation may return, but also a strategy to achieve it. This approach is essentially the step-wise rewriting approach followed, for example, in junior school when teaching children to simplify arithmetic expressions, with the strategy specifying which subexpressions may be simplified at any stage.

Again, configurations are simple arithmetic expressions: $\mathcal{S} = \mathit{Env} \times \mathit{Exp}$ and $\mathcal{V} = \mathit{Num}$. The small-step relation $\longrightarrow_1^e \subseteq \mathit{Env} \times \mathit{Exp} \times \mathit{Exp}$, is between two expressions, given an environment. The important difference with big-step semantics us that expressions do not simplify "in one go" to a value, but rather simplify one step at a time to other expressions, and perhaps finally to values. The reduction relation is also defined inductively, using inference rules, which are *syntax-directed*, but in a sense slightly different from that in the big-step semantics. There may be several rules for the same syntactic construct, and some constructs may have no associated rules. Moreover, the case analysis is not strictly on syntactic structure but rather on an analysis of where in an expression simplification can take place. Small-step reduction relations are seldom transitive and are usually irreflexive.

Table 2 displays a reduction relation for evaluating simple arithmetic expressions. The rule (vbl) says that variables are simplified to the value specified in the given environment. As expected, the rule has a proviso requiring that the variable be in the environment's domain. Note there is no rule for numerals! The rule (add_0) can be understood as saying "$(n_1 + n_2)$ simplifies to the result of $\mathit{ADD}(n_1, n_2)$". The rules (add_l) and (add_r) are symmetric; the former says that if e_1 can simplify to e_1', then $(e_1 + e_2)$ can simplify to $(e_1' + e_2)$ in a single step (similarly for simplifying e_2 first). Note that the relation is non-deterministic, and involves localized rewriting.

Observe that it is possible for an expression, such as $((7 + 21) + y)$ where $y \notin \mathit{dom}(\gamma)$ for a given environment γ, to be reduced a few steps before it gets "stuck". This is in contrast to the big-step situation where no transition is possible for that expression with respect to such an environment γ.

An expression of the form $(n_1 + n_2)$, an instance of the left side in an

$$(vbl) \quad \frac{}{\gamma \vdash x \longrightarrow_1^e \gamma(x)} \qquad \text{provided } x \in dom(\gamma).$$

$$(add_0) \quad \frac{}{\gamma \vdash (n_1 + n_2) \longrightarrow_1^e n_3} \qquad \text{where } n_3 = ADD(n_1, n_2)$$

$$(add_l) \quad \frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash (e_1 + e_2) \longrightarrow_1^e (e_1' + e_2)}$$

$$(add_r) \quad \frac{\gamma \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash (e_1 + e_2) \longrightarrow_1^e (e_1 + e_2')}$$

Table 2: "Small-step" semantics for arithmetic expressions

axiom, is called a *redex*. Any *reducible* expression can be shown to contain a redex. Different small-step relations may be proposed that differ in which redex should be selected first for reduction.

Typical results about small-step semantics usually pertain to the reflexive transitive closure of the reduction relation. For instance, we can show the agreement with the big-step semantics:

Proposition 2.9 *For all* $e \in Exp$, $\gamma \in Env$ *and* $n \in Num$: $\gamma \vdash e \ (\longrightarrow_1^e)^* \ n$ *if and only if* $\gamma \vdash e \Longrightarrow_e n$.

This and similar results are proven by induction on the number of reduction steps involved in $\gamma \vdash e \ (\longrightarrow_1^e)^* \ n$, and within each reduction step, by an induction on the depth of the proof tree justifying the single reduction step.

A corollary to the proposition above is that the "reduction-down-to-values" relation is a (partial) function, though such results can be shown from first principles without reference to the big-step semantics.

A more interesting result to show about the relation $\longrightarrow_1^e$ is whether it satisfies a *strong diamond property*. The proof of this property is by structural induction on the original expression, and analysis on how it could reduce to different expressions using induction on these justifications. This confluence result provides a direct proof that while reduction is non-deterministic, the input-output relation it induces is a function. (Totality is often shown by proving that a reduction relation is well-ordered.)

A confluence result can greatly simplify reasoning about program execution, since it essentially says that we need not consider each possible sequence but merely any one sequence to a point of confluence. Confluence properties can play an important role in compilation, since confluent systems admit simplifications in any order, including strategies that involve simplification of subexpressions in parallel or even in non-deterministic fashion; these may make sense in certain architectures such as those involving pipelining or multiple computational units. Non-confluence should alert a compiler developer that a proposed optimization may in fact be unsound if it alters reduction order, and ought therefore be avoided.

The small-step framework admits various restricted versions of reduction corresponding to specialized strategies, typically those that are deterministic or easier to implement. For instance, we could replace the (add_r) rule by more restrictive versions, *e.g.*,

$$(add_r^{lseq}) \qquad \frac{\gamma \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash (n + e_2) \longrightarrow_1^e (n + e_2')}$$

which allow simplification of the second summand only when the first is already a numeral. With these more restrictive rules, the reduction relation becomes deterministic; for any expression at most one reduction rule applies. The modified relation specifies a sequential left-to-right evaluation strategy. It is then important to prove that this strategy can *simulate* the original relation correctly in the sense that both relations have the same reflexive transitive closures when considering reductions down to values. This result is an example of *standardization*: if an expression can be reduced to a value by any strategy, it can be reduced by a standard sequence using a particular strategy[2].

Standardization is useful in reasoning about program execution, since it allows one to transform any sequence of reductions to another one about which it is somehow easier to reason. Standardization results are often employed, for instance, in showing that certain reduction sequences are not possible. They can be important to a compiler writer, since they permit the use of possibly more efficient implementation strategies without having to sacrifice any generality. It must be emphasized that standardization is a very important syntactic meta-theorem of transition systems that applies only in systems whose extensional behavior (input-output) is deterministic.

[2]Richer languages may require more complicated standardization results.

Example 2.10 *Phenomena such as non-termination sharpen the differences between various evaluation strategies. Consider a simple language of possibly non-terminating boolean expressions given by the abstract grammar:*

$$b := tv \mid \Omega \mid (b_1 \bigvee b_2) \qquad tv \in \{\mathbf{true}, \mathbf{false}\}$$

We define three different small-step relations (omitting the "$\gamma \vdash$" in the rules): $\longrightarrow_1^{comp}$ *which evaluates all parts of a disjunctive boolean expression,*

$$\frac{}{\Omega \longrightarrow_1^{comp} \Omega} \qquad \frac{}{(tv_1 \bigvee tv_2) \longrightarrow_1^{comp} tv_3} \quad tv_3 = OR(tv_1, tv_2)$$

$$\frac{b_1 \longrightarrow_1^{comp} b_1'}{(b_1 \bigvee b_2) \longrightarrow_1^{comp} (b_1' \bigvee b_2)} \qquad \frac{b_2 \longrightarrow_1^{comp} b_2'}{(tv \bigvee b_2) \longrightarrow_1^{comp} (tv \bigvee b_2')}$$

$\longrightarrow_1^{ls}$ *which is a left sequential evaluation,*

$$\frac{}{\Omega \longrightarrow_1^{ls} \Omega} \qquad \frac{b_1 \longrightarrow_1^{ls} b_1'}{(b_1 \bigvee b_2) \longrightarrow_1^{ls} (b_1' \bigvee b_2)}$$

$$\frac{}{(\mathbf{true} \bigvee b) \longrightarrow_1^{ls} \mathbf{true}} \qquad \frac{}{(\mathbf{false} \bigvee b_2) \longrightarrow_1^{ls} b_2}$$

and $\longrightarrow_1^{par}$ *which is parallel evaluation*

$$\frac{}{\Omega \longrightarrow_1^{par} \Omega}$$

$$\frac{b_1 \longrightarrow_1^{par} b_1'}{(b_1 \bigvee b_2) \longrightarrow_1^{par} (b_1' \bigvee b_2)} \qquad \frac{b_2 \longrightarrow_1^{par} b_2'}{(b_1 \bigvee b_2) \longrightarrow_1^{par} (b_1 \bigvee b_2')}$$

$$\frac{}{(b_1 \bigvee \mathbf{false}) \longrightarrow_1^{par} b_1} \qquad \frac{}{(\mathbf{false} \bigvee b_2) \longrightarrow_1^{par} b_2}$$

$$\frac{}{(b_1 \bigvee \mathbf{true}) \longrightarrow_1^{par} \mathbf{true}} \qquad \frac{}{(\mathbf{true} \bigvee b_2) \longrightarrow_1^{par} \mathbf{true}}$$

If a boolean expression b reaches normal form via $\longrightarrow_1^{comp}$ *then it reaches the same normal form via* $\longrightarrow_1^{ls}$*, in which case it reaches the same normal form via* $\longrightarrow_1^{par}$*. However, the converse is not true:*
$(\mathbf{true} \bigvee \Omega) \longrightarrow_1^{par} \mathbf{true}$ *and* $(\Omega \bigvee \mathbf{true}) \longrightarrow_1^{par} \mathbf{true}$*, but*
$(\mathbf{true} \bigvee \Omega) \longrightarrow_1^{ls} \mathbf{true}$ *whereas* $(\Omega \bigvee \mathbf{true}) \longrightarrow_1^{ls} (\Omega \bigvee \mathbf{true})$*. However, both* $(\mathbf{true} \bigvee \Omega) \longrightarrow_1^{comp} (\mathbf{true} \bigvee \Omega)$ *and* $(\Omega \bigvee \mathbf{true}) \longrightarrow_1^{comp} (\Omega \bigvee \mathbf{true})$.

Environment-free formulations. We pause briefly to remark that the formulation of the above relations using environments can be transformed to transition systems that operates wholly within syntax. For this we need the notion of substitution.

Definition 2.11 (substitution) *A substitution σ is a finite domain function from $\mathcal{X}$ to Exp. Equivalently, it can be viewed as a total function that is almost everywhere identity. We write $e\sigma$ to denote applying σ to e yielding an expression obtained by simultaneously replacing in e every occurrence of variable x by the expression $\sigma(x)$ for each $x \in vars(e)$.*

An environment γ is a specific instance of a substitution. It can easily be shown that if $\gamma \vdash e \ (\longrightarrow_1^e)^* \ n$ then $\vdash e\gamma \ (\longrightarrow_1^e)^* \ n$ (the variable free case) and likewise for $\Longrightarrow_e$.

This observation may cause you to wonder why we introduced environments in the first place. The reason is that substitution is usually an expensive operation, whereas the environment data structure allows the computation to "look up" the expression to be substituted for a variable as and when it is needed. Moreover, the later sections will show that environments arise naturally when we try to implement languages with block structure and functions. The environment-less formulation eases the presentation of the following notion of equality.

Operational notions of equality. Given a small-step relation such as $\longrightarrow_1^e$, it is often natural to define a notion of equality $=^e$ on expressions as the *symmetric* reflexive transitive closure of the reduction relation. This is precisely the idea taught in junior school to show that two arithmetic expressions are equal.

Definition 2.12 (Equality) *$e \ =^e \ e'$ if there is a sequence of expressions $e_1, \ldots, e_n$ such that $e \equiv e_1$, $e' \equiv e_n$ and for each $i : 1 \leq i \leq n-1$, either $e_i \ \longrightarrow_1^e \ e_{i+1}$ or $e_{i+1} \ \longrightarrow_1^e \ e_i$.*

If the $\longrightarrow_1^e$ relation is weakly confluent, e and e' can be reduced to a common form.

Abstract machines. A more common approach to specifying arithmetic expression evaluation, familiar to most computer scientists after an introductory data structures course, is by using a stack machine. This semantics is at

a lower level than either the big-step or small-step semantics, since it departs from providing a specification of evaluation directly in terms of the source syntax, and also since it employs additional data structures.

The op-codes of the machine are instructions for loading numerical constants, for adding numerals and for looking up bindings of variables. To avoid introducing new symbols, we employ the same symbols for the op-codes of the machine. Let *OpCodes* be defined as sequences (strings) over the symbol +, numerals, variables in $\mathcal{X}$, with the idea that a variable is a look-up operation[3].

$$OpCodes = (Num \bigcup \mathcal{X} \bigcup \{+\})^*$$

Consider now a post-order traversal of the abstract syntax tree of an expression in *Exp*. This is defined as a recursive function $compile : Exp \rightarrow OpCodes$. To enhance readability, we have used ^ to indicate string catenation.

$$\begin{array}{lcl} compile(n) & = & n \\ compile(x) & = & x \\ compile((e_1 + e_2)) & = & compile(e_1)\hat{}compile(e_2)\hat{}+ \end{array}$$

Configurations of the abstract machine are triples consisting of an environment, a "stack" of numerals, and a sequence of op-codes. Table 3 details the initialization and transitions (the relation $\longrightarrow\!\triangleright$) of the abstract machine. Observe that we have presented a (finite) set of possibly conditional rewrite rules in a two-dimensional syntax. The rules are operated by taking any configuration that matches via a substitution for the schematic variables, *e.g.*, $\gamma, c, S, n, \ldots$, the pattern indicated in the left side of a rule, and replacing it with the configuration obtained by applying the same substitution to the right side of a rule. In this example the rewrite rules involved are deterministic and "regular", in that at most one rule applies and that no configuration can be rewritten to more than one configuration.

The machine is initialized with a given environment γ with respect to which expression e is to be evaluated, an empty stack, and a sequence of op-codes corresponding to $compile(e)$. (For readability we have used the ML-like notation :: for sequence concatenation, writing *e.g.*, $+ :: C'$ to specify a sequence beginning with + followed by sequence C'.) Observe that there are no inference rules — merely rewrite rules, which are applied repeatedly

[3]In implementations, we can have a single op-code that is parametrized by a variable (or equivalently an address or index corresponding to the variable), and similarly a single op-code for loading constants.

$$load(\gamma, e) \;=\; \langle \gamma,\ \lfloor\ \ \rfloor,\ compile(e)\rangle$$

variables	$\langle \gamma,\ S,\ x :: C\rangle \longrightarrow\!\triangleright\ \langle \gamma,\ \left\lfloor \begin{array}{c} \gamma(x) \\ S \end{array} \right\rfloor,\ C\rangle$	
constants	$\langle \gamma,\ S,\ n :: C\rangle \longrightarrow\!\triangleright\ \langle \gamma,\ \left\lfloor \begin{array}{c} n \\ S \end{array} \right\rfloor,\ C\rangle$	
add	$\langle \gamma,\ \left\lfloor \begin{array}{c} n_2 \\ n_1 \\ S \end{array} \right\rfloor,\ + :: C\rangle \longrightarrow\!\triangleright\ \langle \gamma,\ \left\lfloor \begin{array}{c} n_3 \\ S \end{array} \right\rfloor,\ C\rangle$	where $n_3 = ADD(n_1, n_2)$

$$unload(\langle \gamma,\ \lfloor\ n\ \rfloor,\ \epsilon\rangle) = n$$

Table 3: Evaluating expressions using an abstract stack machine

until no rule applies. The moves depend primarily on the first op-code in the sequence. The "good" terminal states are those with a single value on the stack, from which the results are "unloaded", and an empty sequence of op-codes.

The operational semantics induced by the abstract machine is exactly the same as the big-step $\Longrightarrow_e$ (and thus also the closure of the small-step relation).

Proposition 2.13 *For all $e \in Exp$, $\gamma \in Env$ and $n \in Num$: $\gamma \vdash e \Longrightarrow_e n$ if and only if there exists a configuration s such that $load(\gamma, e)$ $(\longrightarrow\!\triangleright)^*$ s and $unload(s) = n$*

The proof involves induction on e and on the number of $\longrightarrow\!\triangleright$ steps in reaching the terminal configuration. In fact, several non-trivial lemmas need to be shown, which essentially state that the evaluation of an expression does not examine or disturb the part of the stack below its initial top, and that any expression results in a single value on the stack.

A typical result that has to be shown is along the lines of "for *any* stack S and code list C', if

$$\langle \gamma,\ \lfloor\ S\ \rfloor,\ compile(e)\hat{}C'\rangle\ (\longrightarrow\!\triangleright)^*\ \langle \gamma,\ \lfloor\ S'\ \rfloor,\ C'\rangle$$

then $\lfloor\ S'\ \rfloor = \left\lfloor \begin{array}{c} n \\ S \end{array} \right\rfloor$ for some $n \in Num$." The proof is by induction on the length of the op-code sequence, but observe that we need to explicitly involve all "contexts" in which an expression may be evaluated — the universal quantification on all stacks S and "continuation" code C' — in the statement of this property.

The above abstract machine can be seen as an implementation of a left-to-right reduction. In general, standardization results help mediate the relationship between the abstract machine semantics and the reduction semantics.

Tuples, records and conditionals. We make a quick foray into giving rules for structured expressions. We consider pairs (the idea extends easily to tuples and records) and a simple conditional (which generalizes to **case** statements. We only point out that in the rules for conditionals, the test e_1 is *first* evaluated to a truth value before *one* of the branches e_2 or e_3 is selected.

We assume that our values $v ::= n \mid tv \mid \langle v_1,\ v_2 \rangle$. The big-step rules for pairs and conditionals are:

$$(pair)\quad \frac{\gamma\ \vdash\ e_1\ \Longrightarrow_e\ v_1 \quad \gamma\ \vdash\ e_2\ \Longrightarrow_e\ v_2}{\gamma\ \vdash\ \langle e_1,\ e_2\rangle\ \Longrightarrow_e\ \langle v_1,\ v_2\rangle}$$

$$(if_t)\quad \frac{\gamma\ \vdash\ e_1\ \Longrightarrow_e\ \mathbf{true} \quad \gamma\ \vdash\ e_2\ \Longrightarrow_e\ v_2}{\gamma\ \vdash\ \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \Longrightarrow_e\ v_2}$$

$$(if_f)\quad \frac{\gamma\ \vdash\ e_1\ \Longrightarrow_e\ \mathbf{false} \quad \gamma\ \vdash\ e_3\ \Longrightarrow_e\ v_3}{\gamma\ \vdash\ \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \Longrightarrow_e\ v_3}$$

One possible set of small step rules are:

$$(pair_l)\quad \frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash \langle e_1,\ e_2\rangle \longrightarrow_1^e \langle e_1',\ e_2\rangle}$$

$$(pair_r)\quad \frac{\gamma \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash \langle e_1,\ e_2\rangle \longrightarrow_1^e \langle e_1,\ e_2'\rangle}$$

$$(if_0)\quad \frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \longrightarrow_1^e \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

$$(if_l)\quad \frac{}{\gamma \vdash \textbf{if true then } e_2 \textbf{ else } e_3 \longrightarrow_1^e e_2}$$

$$(if_r)\quad \frac{}{\gamma \vdash \textbf{if false then } e_2 \textbf{ else } e_3 \longrightarrow_1^e e_3}$$

We do not present the abstract machine rules but observe that new op-codes need to be introduced, and the *compile* function extended. The rule for a n-tuple-formation op-code takes n values off the stack forms an n-tuple which is then pushed onto the stack. A record formation operation will require a little more jugglery, for example, by sorting the fields according a particular order (lexicographic, say) at the compilation stage, and traversing the abstract syntax tree accordingly. The op-code for conditional choice picks one of two *continuations* based on the value on the top of the stack. At an abstract level, it is possible to talk of compound op-codes $IF(c_1, c_2)$, which are realized on actual machines by jumps. Another trick, used by Plotkin [Plo81, page 18] as a motivating illustration for advocating more structure in operational descriptions, is to take the syntax apart and stash the continuations or markers, selecting the correct one based on the evaluation of e_1.

2.3 Private definitions

Tennent's *principle of qualification* [Ten81] suggests that *Exp* can be extended to include expressions that employ locally scoped definitions.

$$e \ ::= \ \ldots \mid \textbf{let } x \stackrel{def}{=} e_1 \textbf{ in } e_2$$

In $\textbf{let } x \stackrel{def}{=} e_1 \textbf{ in } e_2$, the *scope* of the definition of x to e_1 is limited to e_2. The occurrences of variables in an expression are now of two kinds: those which

are bound and those which are free. Define $fv : \mathit{Exp} \to \mathcal{X}$ as:

$$fv(x) = x \quad fv(f(e_1, ..., e_k)) = \bigcup_{i=i}^{k} fv(e_i)$$
$$fv(c) = \emptyset \quad fv(\textbf{let } x \stackrel{def}{=} e_1 \textbf{ in } e_2) = fv(e_1) \bigcup (fv(e_2) - \{x\})$$

The big-step rules are extended with:

$$\frac{\gamma \vdash e_1 \Longrightarrow_e n_1 \quad \gamma[x \mapsto n_1] \vdash e_2 \Longrightarrow_e n_2}{\gamma \vdash \textbf{let } x \stackrel{def}{=} e_1 \textbf{ in } e_2 \Longrightarrow_e n_2}$$

The small-step rules are:

$$\frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash \textbf{let } x \stackrel{def}{=} e_1 \textbf{ in } e_2 \longrightarrow_1^e \textbf{let } x \stackrel{def}{=} e_1' \textbf{ in } e_2}$$

$$\frac{\gamma[x \mapsto n_1] \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash \textbf{let } x \stackrel{def}{=} n_1 \textbf{ in } e_2 \longrightarrow_1^e \textbf{let } x \stackrel{def}{=} n_1 \textbf{ in } e_2'}$$

$$\frac{}{\gamma \vdash \textbf{let } x \stackrel{def}{=} n_1 \textbf{ in } n_2 \longrightarrow_1^e n_2}$$

We postpone the presentation of an abstract machine that can correctly deal with scoping issues to our discussion of functions in §4, since the machinery needed there subsumes that needed here. Tennent's *principle of correspondence* relates definition mechanisms to parameter-passing and thus definition mechanisms get addressed in the operational semantics for function definition and call. It suffices to mention at this stage that the machines will now additionally have to stack environments (or structures containing them) to implement the lexical scoping of block structured languages.

Instead we will discuss compound definitions. Consider the syntactic category *Defs* with meta-variable d:

$$d \ ::= \ x \stackrel{def}{=} e \mid d_1;d_2 \mid d_1 \| d_2$$

with $dv : \mathit{Defs} \to \mathcal{X}$ returning the defined variables, and fv extended to *Defs*

$$dv(x \stackrel{def}{=} e) = \{x\} \quad dv(d_1;d_2) = dv(d_1) \bigcup dv(d_2) \quad dv(d_1 \| d_2) = dv(d_1) \uplus dv(d_2)$$

$$fv(x \stackrel{def}{=} e) = fv(e) \quad fv(d_1;d_2) = fv(d_1) \bigcup (fv(d_2) - dv(d_1))$$
$$fv(d_1 \| d_2) = fv(d_1) \bigcup fv(d_2)$$

Here $\uplus$ stands for disjoint union, defined only when the sets are actually disjoint.

The big-step semantics uses two mutually recursive (but nevertheless inductive definitions): $\Longrightarrow_e$ as before and $\Longrightarrow_d \subseteq Env \times Defs \times Env$. Observe here that the "values" (canonical forms) for the $\Longrightarrow_d$ transition system are *environments*, which are "extra-syntactic"! The rules for $\Longrightarrow_d$ are:

$$\frac{\gamma \vdash e \Longrightarrow_e n}{\gamma \vdash x \stackrel{def}{=} e \Longrightarrow_d [x \mapsto n]}$$

$$\frac{\gamma \vdash d_1 \Longrightarrow_d \gamma_1 \quad \gamma[\gamma_1] \vdash d_2 \Longrightarrow_d \gamma_2}{\gamma \vdash d_1;d_2 \Longrightarrow_d \gamma_1[\gamma_2]}$$

$$\frac{\gamma \vdash d_1 \Longrightarrow_d \gamma_1 \quad \gamma \vdash d_2 \Longrightarrow_d \gamma_2}{\gamma \vdash d_1 \| d_2 \Longrightarrow_d \gamma_1 \bigcup \gamma_2}$$

To correctly implement scoping, $\Longrightarrow_d$ returns the incremental change to the environment obtained by processing a definition. In sequential definitions we first process d_1 with respect to γ, which we augment with the resulting environment to process d_2, whereas with simultaneous definitions, the same environment is used for elaborating the parallel definitions. In the last rule, since we assumed that $dv(d_1) \bigcap dv(d_2) = \emptyset$, the union of environments is well-defined.

Finally, since the principle of qualification may be applied to *Defs*, we obtain definitions that contain auxiliary local definitions

$$d ::= \ldots \mid \textbf{local}\ d_1\ \textbf{in}\ d_2$$

$$dv(\textbf{local}\ d_1\ \textbf{in}\ d_2) = dv(d_2) \quad fv(\textbf{local}\ d_1\ \textbf{in}\ d_2) = fv(d_1) \bigcup (fv(d_2) - dv(d_1))$$

The big-step semantics of this construct is:

$$\frac{\gamma \vdash d_1 \Longrightarrow_d \gamma_1 \quad \gamma[\gamma_1] \vdash d_2 \Longrightarrow_d \gamma_2}{\gamma \vdash \textbf{local}\ d_1\ \textbf{in}\ d_2 \Longrightarrow_d \gamma_2}$$

The reduction semantics for *Defs* is somewhat more tricky (see [Plo81, pages 80-81]). The problem can perhaps be understood in trying to reduce **let** d **in** e when d is irreducible. Here, the bindings of d must somehow be augmented to the outer environment γ before e can be evaluated. Plotkin

employs the expedient of treating environments as a canonical form of definitions, clarifying that they are not in the abstract syntax but merely in the control component in configurations.

$$\frac{\gamma[\gamma_1] \vdash e \longrightarrow_1^e e'}{\gamma \vdash \textbf{let } \gamma_1 \textbf{ in } e \longrightarrow_1^e \textbf{let } \gamma_1 \textbf{ in } e'}$$

Indeed, this mixing of extra-syntactic data structures (environments) with abstract syntax is a somewhat weak point about pure reduction semantics. While the big-step formulations also use extra-syntactic constructions, their use is far more disciplined (indeed Astesiano points out that various *semantic* definitions can be given in the same inductive framework of big-step semantics) [Ast91].

Relation to types. We finish this section with an important issue of how the operational semantics relates to type-checking. Indeed, our presentation has avoided typing issues altogether, although they are a significant part of any structural semantic presentation. The relationship between typing and execution is particularly significant in strongly-typed languages with compile-time type-checking: Programs that type-check correctly at compile time should not raise *type errors* at run-time. This property can be guaranteed if expressions do not change type during execution. Such a theorem is called *subject reduction*. A typical subject reduction (stated for small-step semantics, but an analogous statement holds for big-step semantics) is:

> Let Γ be a set of assumptions of types of variables under which expression e has type τ (written $\Gamma \vdash e : \tau$). If γ is an environment that conforms to Γ, *i.e.*, it binds variables to values having type according to Γ, and if $\gamma \vdash e \longrightarrow_1^e e'$, then $\Gamma \vdash e' : \tau$.

3 Imperative Languages

We now move on to providing a simple imperative language WHILE with operational semantics. WHILE has nested within it a language of expressions (boolean expressions in particular) and the operational semantics provides a good illustration of how semantics developed for one syntactic category can be employed in the inductive definition of another — transitions for expressions are employed in those for imperative commands.

Syntax. The syntax of commands *Comm* in WHILE with typical metavariable c is given by the following abstract grammar, where metavariable e ranges over *Exp*, which we assume includes a sublanguage of boolean expressions.

$$\begin{array}{rcl} c & ::= & \textbf{skip} \\ & | & x{:=}e \\ & | & c_1;c_2 \\ & | & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \\ & | & \textbf{while } e \textbf{ do } c \end{array}$$

Big-step semantics. The big-step semantics of WHILE, as noted earlier, is a relational specification of command execution. The imperative model of computation is based on the idea of making a series of small changes to a *memory state*. Commands can be thought of as state transformers — the basic action being that of assigning a value to a program variable. More complex actions are built up from the elementary ones using constructs for sequencing, conditional execution and iteration. For convenience, we include an identity transformation, namely the command **skip**.

Let *State* consist of finite domain functions from $\mathcal{X}$ to $\mathcal{V}$. For simplicity, we will assume that expression evaluation involves no "side-effects" that change the state of memory. *State* is, at least as a first approximation, the same as *Env*. This valuable abstraction will get taken apart in modeling other features. The set of configurations in the transition system is $(State \times Comm) \bigcup State$. The big-step transition relation $\Longrightarrow \subseteq (State \times Comm) \times State$ is defined as the smallest relation closed under the rules given in Table 4.

skip leaves the state unchanged. If an expression e evaluates to a value v in a state σ (given in terms of the big-step relation for expressions), the effect of an assignment $x{:=}e$ results in a state that is identical to σ, except that its value at variable x is now v. If c_1 transforms σ to σ_1 and c_2 transforms σ_1 to σ_2, then their sequential composition achieves the composite transformation of σ to σ_2. The rules for the conditional say that **if** e **then** c_1 **else** c_2 transforms σ as would command c_1 (respectively c_2) depending on whether e evaluates to **true** or **false** in state σ. The *while* rules for the indefinite iterator are also intuitive – if the boolean condition e evaluates to **false** in state σ, the loop is not entered; if e evaluates to **true** then if the body c of the loop is executed to reach state σ_1 and if executing the loop **while** e **do** c

$$skip \qquad \frac{}{\langle \sigma,\ \mathbf{skip} \rangle \Longrightarrow \sigma}$$

$$assign \qquad \frac{\sigma \ \vdash \ e \ \Longrightarrow_e \ v}{\langle \sigma,\ x{:=}e \rangle \Longrightarrow \sigma[x \mapsto v]}$$

$$seq \qquad \frac{\langle \sigma,\ c_1 \rangle \Longrightarrow \sigma_1 \quad \langle \sigma_1,\ c_2 \rangle \Longrightarrow \sigma_2}{\langle \sigma,\ c_1;c_2 \rangle \Longrightarrow \sigma_2}$$

$$if_{\,true} \qquad \frac{\sigma \ \vdash \ e \ \Longrightarrow_e \ \mathbf{true} \quad \langle \sigma,\ c_1 \rangle \Longrightarrow \sigma_1}{\langle \sigma,\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rangle \Longrightarrow \sigma_1}$$

$$if_{\,false} \qquad \frac{\sigma \ \vdash \ e \ \Longrightarrow_e \ \mathbf{false} \quad \langle \sigma,\ c_2 \rangle \Longrightarrow \sigma_2}{\langle \sigma,\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rangle \Longrightarrow \sigma_2}$$

$$while_{false} \qquad \frac{\sigma \ \vdash \ e \ \Longrightarrow_e \ \mathbf{false}}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c \rangle \Longrightarrow \sigma}$$

$$while_{\,true} \qquad \frac{\sigma \ \vdash \ e \ \Longrightarrow_e \ \mathbf{true} \quad \langle \sigma,\ c \rangle \Longrightarrow \sigma_1 \quad \langle \sigma_1,\ \mathbf{while}\ e\ \mathbf{do}\ c \rangle \Longrightarrow \sigma_2}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c \rangle \Longrightarrow \sigma_2}$$

Table 4: "Big-step" semantics for a simple imperative language

starting from σ_1 yields state σ_2, then σ_2 is the resulting state from executing the loop. Observe that this relational specification corresponds to *partial correctness*[4].

The $\Longrightarrow$ relation is deterministic:

Proposition 3.1 *If* $\langle \sigma,\ c\rangle \Longrightarrow \sigma_1$ *and* $\langle \sigma,\ c\rangle \Longrightarrow \sigma_2$ *then* $\sigma_1 = \sigma_2$.

There are some subtle technical issues about these rules that arise, *e.g.*, in formal compiler verification exercises. As noted in [Ast91], the rules for the **while** e **do** c yield an inductive definition, but one which is *not structural*. The two *while* rules can be coalesced into a single equivalent rule, which too is not structural:

$$\frac{\langle \sigma,\ \textbf{if } e \textbf{ then } c;\textbf{while } e \textbf{ do } c \textbf{ else skip}\rangle \implies \sigma'}{\langle \sigma,\ \textbf{while } e \textbf{ do } c\rangle \implies \sigma'}$$

Both these formulations involve a recursive definition, which while being concise and intuitive do not allow the use of structural induction. Fortunately, there is an equivalent formulation for the **while** e **do** c rule which is structural; this formulation employs an auxiliary inductively defined relation $\mathcal{F} \subseteq \mathit{State} \times \mathit{State}$.

$$\frac{\langle \sigma,\ \sigma'\rangle \in \mathcal{F}}{\langle \sigma,\ \textbf{while } e \textbf{ do } c\rangle \implies \sigma'}$$

where $\mathcal{F}$ is defined inductively by:

$$\frac{\sigma \vdash e \Longrightarrow_e \textbf{false}}{\langle \sigma,\ \sigma\rangle \in \mathcal{F}}$$

$$\frac{\sigma \vdash e \Longrightarrow_e \textbf{true} \quad \langle \sigma,\ c\rangle \implies \sigma'' \quad \langle \sigma'',\ \sigma'\rangle \in \mathcal{F}}{\langle \sigma,\ \sigma'\rangle \in \mathcal{F}}$$

We can now propose operational notions of equivalence and ordering between While programs:

Definition 3.2 (Operational equivalence)
$c_1 \preceq c_2$ *whenever for all* σ: $\langle \sigma,\ c_1\rangle \Longrightarrow \sigma_1 \supset \langle \sigma,\ c_2\rangle \Longrightarrow \sigma_1$
$c_1 \approx c_2$ *whenever for all* σ: $\langle \sigma,\ c_1\rangle \Longrightarrow \sigma_1$ *if and only if* $\langle \sigma,\ c_2\rangle \Longrightarrow \sigma_1$.

[4] In fact, it is possible to read the Hoare style axiomatic semantics for While as being a "backwards" operational semantics on a non-standard kind of state.

These notions are instances of the concepts of Definition 2.3 — the observable behavior of a command is how it transforms a given state to yield a resulting state.

Example 3.3 *Here are some equivalences and ordering relations that can be seen as code improvements.*

1. **skip** $\approx$ **while false do** c *for all commands* c.
2. **while true do** $c \preceq c'$ *for all* c, c' *since the former is non-terminating.*
3. *Let* $W \equiv$ **while** e **do** c. *Then* $W \approx$ **if** e **then** c;W **else skip**.
4. c;**skip** $\approx c \approx$ **skip**;c *for all* c.
5. **if true then** c_1 **else** $c_2 \approx c_1$ *and* **if false then** c_1 **else** $c_2 \approx c_2$.

Reduction semantics. We now move on to the reduction semantics for WHILE as a possibly more detailed description on how to realize an implementation. The main difference now is the relation $\longrightarrow_1 \subseteq (State \times Comm) \times ((State \times Comm) \bigcup State)$. The canonical (normal) forms for this relation are, naturally, those in $State$. Table 5 presents the reduction semantics.

The $\longrightarrow_1$ relation is easy to understand. Rule *skip′* says **skip** does nothing. Executing an assignment first involves simplifying the expression e (repeatedly using rule $assign'_1$) down to a value v, which is then associated with x (rule $assign'_2$). Executing a sequential composition $c_1;c_2$ involves executing the first command c_1 until it is exhausted (repeatedly using rule seq'_1), at which stage we start the execution of c_2 from the resulting state σ_1 (rule seq'_2). In evaluating a conditional, we first evaluate the expression e to a boolean value (repeatedly using rule if'_1). If that value is **true**, then c_1 is executed (rule if'_{true}) and if it is **false**, c_2 is executed (rule if'_{false}). The *while′* rule is, again, somewhat harder to formalize concisely, and relies on the fact that **skip** is a "no op".

Proposition 3.4 *The big-step and reduction semantics define the same notion of program execution, that is, for all* c *and* σ: $\langle \sigma,\ c \rangle \implies \sigma'$ *if and only if* $\langle \sigma,\ c \rangle\ (\longrightarrow_1)^*\ \sigma'$.

$$skip' \quad \frac{}{\langle \sigma,\ \mathbf{skip} \rangle \ \longrightarrow_1 \ \sigma}$$

$$assign'_1 \quad \frac{\sigma \ \vdash \ e \ \longrightarrow_1^e \ e'}{\langle \sigma,\ x{:=}e \rangle \ \longrightarrow_1 \ \langle \sigma,\ x{:=}e' \rangle}$$

$$assign'_2 \quad \frac{}{\langle \sigma,\ x{:=}v \rangle \ \longrightarrow_1 \ \sigma[x \mapsto v]}$$

$$seq'_1 \quad \frac{\langle \sigma,\ c_1 \rangle \ \longrightarrow_1 \ \langle \sigma_1,\ c_1' \rangle}{\langle \sigma,\ c_1;c_2 \rangle \ \longrightarrow_1 \ \langle \sigma_1,\ c_1';c_2 \rangle}$$

$$seq'_2 \quad \frac{\langle \sigma,\ c_1 \rangle \ \longrightarrow_1 \ \sigma_1}{\langle \sigma,\ c_1;c_2 \rangle \ \longrightarrow_1 \ \langle \sigma_1,\ c_2 \rangle}$$

$$if'_1 \quad \frac{\sigma \ \vdash \ e \ \longrightarrow_1^e \ e'}{\langle \sigma,\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rangle \ \longrightarrow_1 \ \langle \sigma,\ \mathbf{if}\ e'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \rangle}$$

$$if'_{true} \quad \frac{}{\langle \sigma,\ \mathbf{if\ true\ then}\ c_1\ \mathbf{else}\ c_2 \rangle \ \longrightarrow_1 \ \langle \sigma,\ c_1 \rangle}$$

$$if'_{false} \quad \frac{}{\langle \sigma,\ \mathbf{if\ false\ then}\ c_1\ \mathbf{else}\ c_2 \rangle \ \longrightarrow_1 \ \langle \sigma,\ c_2 \rangle}$$

$$while' \quad \frac{\langle \sigma,\ \mathbf{if}\ e\ \mathbf{then}\ c;\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{else\ skip} \rangle \ \longrightarrow_1 \ \langle \sigma',\ c' \rangle}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c \rangle \ \longrightarrow_1 \ \langle \sigma',\ c' \rangle}$$

Table 5: Reduction semantics for a simple imperative language

Remark 3.5 *In fact, in [Plo81], Plotkin uses what Astesiano calls a "mixed step" semantics for branching and iteration. For example, the rules for while he gives are:*

$$\frac{\sigma \vdash e\ (\longrightarrow_1^e)^*\ \mathbf{true}}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c\rangle \longrightarrow_1 \langle \sigma,\ c;\mathbf{while}\ e\ \mathbf{do}\ c\rangle} \qquad \frac{\sigma \vdash e\ (\longrightarrow_1^e)^*\ \mathbf{false}}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c\rangle \longrightarrow_1 \sigma}$$

His small step rules for the while command involve the transitive closure of the small step reduction of expressions, equivalent to a big step expression evaluation.

Recall that a small-step semantics can be thought of as moving "irrevocably forward" albeit non-deterministically, whereas big-step semantics can easily incorporate temporary undo-able changes in describing sub-computations. Constructs which have a relatively simple big-step semantics but have difficult small-step semantics usually necessitate additional data structures such as stacks for effecting the temporary changes involved in sub-computations in the abstract machine.

Abstract machine. The abstract machine for WHILE is a so-called SMC machine, with a *S*tack for evaluating expressions, a *M*emory or *State* component and a *C*ode list. As illustrated by Plotkin [Plo81, pages 17-19], the transition semantics is somewhat messy: the transition relation is *not* directly in terms of syntactic structure, and the linearization of this abstract syntax via a post-order traversal that worked well for expressions requires "adjustments" for constructs involving branching and iteration, wherein control points need to be stacked for further use or disposal. The reason is that execution of a program is no longer isomorphic to traversal of the abstract syntax tree, since a transition sequence can involve executing constructs in which an entire subtree may be ignored (branching and iteration), or may be revisited repeatedly (iteration).

3.1 Non-determinism

Dijkstra's so-called guarded choice language is a quintessential imperative language involving non-determinism.

$$c\ ::=\ \ldots \mid \mathbf{if} \bigsqcup_{i=1}^{n} e_i \triangleright c_i\ \mathbf{fi} \mid \mathbf{do} \bigsqcup_{i=1}^{n} e_i \triangleright c_i\ \mathbf{od}$$

Given below are the big-step and mixed step semantics for the new constructs. We use the mixed step approach of Plotkin (or Astesiano) for reduction, since

it yields a compact presentation (a pure small-step presentation is replete with the problems mentioned above). The big-step rules are:

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true} \quad \langle \sigma,\ c_j \rangle \Longrightarrow \sigma'}{\langle \sigma,\ \mathbf{if} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{fi} \rangle \Longrightarrow \sigma'} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true} \quad \langle \sigma,\ c_j ; \mathbf{do} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{od} \rangle \Longrightarrow \sigma'}{\langle \sigma,\ \mathbf{do} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{od} \rangle \Longrightarrow \sigma'} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\bigwedge_{i=1}^{n} \sigma \vdash e_i \Longrightarrow_e \mathbf{false}}{\langle \sigma,\ \mathbf{do} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{od} \rangle \Longrightarrow \sigma}$$

and in the mixed-step formulation:

$$\frac{\sigma \vdash e_j\ (\longrightarrow_1^e)^*\ \mathbf{true}}{\langle \sigma,\ \mathbf{if} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{fi} \rangle \longrightarrow_1 \langle \sigma,\ c_j \rangle} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\sigma \vdash e_j\ (\longrightarrow_1^e)^*\ \mathbf{true}}{\langle \sigma,\ \mathbf{do} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{od} \rangle \longrightarrow_1 \langle \sigma,\ c_j ; \mathbf{do} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{od} \rangle} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\bigwedge_{i=1}^{n} \sigma \vdash e_i\ (\longrightarrow_1^e)^*\ \mathbf{false}}{\langle \sigma,\ \mathbf{do} \ \Box_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{od} \rangle \longrightarrow_1 \sigma}$$

In each set, the first two rules are really families of rules (one for each choice of j). The rule for guarded choice says that if any e_i evaluates to true in σ, then the corresponding c_i may be executed from σ. The rules for guarded iteration say that if any e_i evaluates to true in σ, then the corresponding c_i followed by the loop again may be executed from σ, whereas if all e_i evaluate to false, control exits the iteration construct.

An implementation (or abstract machine) will have to use some mechanism for evaluating the guard expressions down to values choosing some order. In order to achieve some degree of fairness, a scheduler may be used to select the order in which guard expressions will be tried.

Parallel execution. Many concurrent imperative languages allow parallel execution of threads, for instance in a **cobegin-coend** construct.

$$c \ ::= \ \ldots \mid c_1 \| c_2$$

Consider the following big-step semantics:

$$\frac{\langle \sigma,\ c_1 \rangle \Longrightarrow \sigma_1 \quad \langle \sigma,\ c_2 \rangle \Longrightarrow \sigma_2}{\langle \sigma,\ c_1 \| c_2 \rangle \Longrightarrow \sigma_1 \bigcup \sigma_2} \quad \text{provided } W(c_1) \bigcap W(c_2) = \emptyset.$$

where $W(c_i)$ denotes the set of variables changed in c_i. The proviso ensures that the union is well-defined. Unfortunately, this semantics does not correspond to our usual intuition of parallel computation. It is intuitive and simple only when neither thread uses the contents of variables modified by the other (Bernstein's conditions); otherwise it is difficult to implement.

The small-step semantics is simpler to implement (and less fussy to specify!).

$$\frac{\langle \sigma,\ c_i \rangle \longrightarrow_1 \langle \sigma',\ c_i' \rangle}{\langle \sigma,\ c_1 \| c_2 \rangle \longrightarrow_1 \langle \sigma',\ c_1' \| c_2' \rangle} \quad i \in \{1, 2\}.$$

$$\frac{\langle \sigma,\ c_i \rangle \longrightarrow_1 \sigma'}{\langle \sigma,\ c_1 \| c_2 \rangle \longrightarrow_1 \langle \sigma',\ c_{3-i} \rangle} \quad i \in \{1, 2\}.$$

What this suggests is that the granularity of abstraction that the big-step semantics seeks to impose in describing the operational behavior is inappropriate for concurrent computation. Also, using big-step semantics makes it difficult to describe visible side-effects of a computation during non-terminating runs. Consequently, it is common to find most frameworks for concurrency, *e.g.*, [Mil80], using (generally labelled) reduction semantics.

3.2 Blocks and Variable Declarations

Imperative languages are block-structured, and employ scoped declarations of "variables", which are (often) initialized before any command is executed. Moreover, we have not studied any constructs where imperative "variables" (which are really *named storage cells*) can have any structure. A more general treatment of imperative variables is to factor the notion of *State* into two maps, the first an environment $\gamma \in \mathit{Env} = \mathcal{X} \rightarrow_{\mathit{fin}} \mathit{Loc}$ and the second $\sigma \in \mathit{Store} = \mathit{Loc} \rightarrow_{\mathit{fin}} \mathcal{V}$, where *Loc* is a set of storage addresses or *locations* and $\mathcal{V}$ is the set of (storable) values. Environments can also be used to model *constant declarations* by including $\mathcal{V}$ in the co-domain of *Env*. The common practice is to have different environment components for constants, variables, procedures, types, classes, modules — whatever distinct nameable concepts

appear in the language. In what follows, we will assume that the appropriate environment component is being looked up.

We will ignore the issue of the types of the declared variables, since they are (usually) irrelevant for specifying the dynamic behavior. Consequently, variable declarations merely become lists of variables.

$$c \ ::= \ \ldots \mid \textbf{var}\ vd\ \textbf{begin}\ c\ \textbf{end} \qquad vd ::= x \mid vd;vd$$

The big-step relations now are $\Longrightarrow_e \subseteq ((Env \times Store) \times Exp) \times \mathcal{V}$ for expressions, $\Longrightarrow_c \subseteq Env \times (Store \times Comm) \times Store$ for commands, $\Longrightarrow_d \subseteq (Env \times Store \times Decl) \times (Env \times Store)$ for declarations (this is a little more general than we need but will allow variable initializations during declarations, and will be invaluable in the specification of procedures).

The previous rules for expressions will now be relative to a pair γ, σ, and other than the variable lookup all other rules are otherwise unchanged. All the previous rules given for commands will now be relative to an environment γ, and $\Longrightarrow$ will be subscripted $\Longrightarrow_c$. We now give the new and changed rules for variable lookup variable declarations, assignments and blocks (only for the big-step case — we encounter the same issues in blocks as we did in local declarations when attempting a small-step formalization)

$$\frac{}{\gamma, \sigma \ \vdash \ x \ \Longrightarrow_e \ v} \quad \text{where } v = \sigma(\gamma(x)), \text{ if defined}$$

$$\frac{\gamma, \sigma \ \vdash \ e \ \Longrightarrow_e \ v}{\gamma \ \vdash \ \langle \sigma, \ x{:=}e \rangle \Longrightarrow_c \sigma[\gamma(x) \mapsto v]} \quad \text{provided } x \in dom(\gamma).$$

$$\frac{}{\langle \gamma, \ \sigma, \ x \rangle \Longrightarrow_d \langle [x \mapsto l], \ \sigma[l \mapsto \bot] \rangle} \quad \text{where } l \notin codom(\gamma) \bigcup dom(\sigma)$$

$$\frac{\langle \gamma, \ \sigma, \ vd_1 \rangle \Longrightarrow_d \langle \gamma_1, \ \sigma_1 \rangle \quad \langle \gamma[\gamma_1], \ \sigma_1, \ vd_2 \rangle \Longrightarrow_d \langle \gamma_2, \ \sigma_2 \rangle}{\langle \gamma, \ \sigma, \ vd_1;vd_2 \rangle \Longrightarrow_d \langle \gamma_1[\gamma_2], \ \sigma_2 \rangle}$$

$$\frac{\langle \gamma, \ \sigma, \ vd \rangle \Longrightarrow_d \langle \gamma_1, \ \sigma_1 \rangle \quad \gamma[\gamma_1] \ \vdash \langle \sigma_1, \ c \rangle \Longrightarrow_c \sigma'}{\gamma \ \vdash \ \langle \sigma, \ \textbf{var}\ vd\ \textbf{begin}\ c\ \textbf{end} \rangle \Longrightarrow_c \sigma' \wr dom(\sigma)}$$

In assignments, we now use γ to determine the location corresponding to x, which is updated in the store. In variable declarations, fresh locations are generated, and added to the store (initialized to an "undefined value" $\bot$), and then bound to the variables in the environment. Observe that we have (somewhat idiosyncratically) the environments returned be increments

(and hence undo-able), whereas the changes to the store be cumulative (*i.e.*, persistent). This approach is appropriate for small or mixed step semantics, and also for any extensions to procedures. At the abstract machine level, this hints that environments must necessarily to be implemented using stacks (whereas stores can be global, with careful control on accessibility of locations).

Note also that the returned state in the execution of a block purges all components of the store that were created during execution of the block. This is to avoid the occurrence of locations inaccessible from the environment (*i.e.*, garbage). Likewise, we have been careful to avoid the possibility of *dangling references*, namely locations accessible from the environment but not present in the domain of the store — which can occur if we have a command **free**(x).

3.3 Procedures and parameter passing

We now introduce the possibility of declaring and calling procedures in the language While. We consider only non-recursive procedures, with a single variable. The extension to several variables and indeed to several variables with several different parameter-passing mechanisms is at least intuitively straightforward (though rather tedious to write as rules). The extension to recursive procedures, however, is not quite trivial (it involves the computation of fixed points by an iterative process akin to the case of the **while** command).

Indeed, we have met some of the scoping issues during our treatment of blocks (via Tennent's principle of correspondence, where any parameter passing mechanism corresponds to a definition mechanism and conversely). The issues of managing control during call and return are better treated in a more general setting of first-class abstractions in §4.

Parameterless procedures. We first extend the language with facilities for declaring and calling procedures without parameters. It is then easy to extend this further to various parameter-passing conventions such as *call-by-value* and *call-by-reference*.

$$\begin{array}{lcl} d & ::= & \ldots \mid \textbf{sub}\ P\ =\ c \\ c & ::= & \ldots \mid \textbf{call}\ P \end{array}$$

As in most programming languages, we assume that the body c of the procedure **sub** may refer to and modify non-local (free) variables that are visible by the usual rules of static scope.

Semantically a (parameter-less) procedure is merely a state transformer with a name. Hence it is necessary to include state transformers in the co-domain of semantic environments:

$$\begin{array}{lcl} \mathit{Proc}_0 & = & \mathit{Store} \rightarrow_p \mathit{Store} \\ \mathit{Env} & = & \mathcal{X} \rightarrow_{\mathit{fin}} (\mathit{Loc} + \mathit{Proc}_0 + \ldots) \end{array}$$

Operationally, however, a procedure identifier merely represents sufficient information required to be able to execute the code of the procedure. Lexical scoping requires that variables in the body of the procedure take their bindings in the environment that the procedure was declared, rather than from the calling context. Hence a procedure declaration modifies the environment by associating with the procedure name, the environment in which the declaration occurs and the body of the procedure. Such a data structure is called a *procedural closure*. We will revisit closures in §4 while discussing function call in lexically scoped functional languages. As in the case of blocks and declarations, we assume that the state has two components, an environment γ, and a store σ.

$$\mathit{Sub}_0 \quad \frac{}{\langle \gamma,\ \sigma,\ \textbf{sub}\ P\ =\ c\rangle \Longrightarrow_d \langle \gamma[P \mapsto \texttt{proc0}\langle c,\ \gamma\rangle],\ \sigma\rangle}$$

$$\mathit{Call}_0 \quad \frac{\langle \gamma_1,\ \sigma,\ c\rangle \Longrightarrow_c \sigma'}{\langle \gamma,\ \sigma,\ \textbf{call}\ P\rangle \Longrightarrow_c \sigma'} \qquad \gamma(P) = \texttt{proc0}\langle c,\ \gamma_1\rangle$$

Procedures with parameters. Extending the treatment to procedures with parameters, we consider, for simplicity, only procedures with a single parameter. We also consider only the call-by-value and call-by reference mechanisms. The extended language syntax is:

$$\begin{array}{lcl} d & ::= & \ldots \mid \textbf{sub}\ P\ (\textbf{val}\ x) =\ c \mid \textbf{sub}\ P\ (\textbf{var}\ x) =\ c \\ c & ::= & \ldots \mid \textbf{call}\ P(e) \end{array}$$

We require that the expression e can only be a variable symbol when the procedure P uses a **var** parameter. The mathematical domains for procedures of these kinds are:

$$\begin{array}{lcl} \mathit{Proc}_v & = & (\mathit{Store} \times \mathcal{V}) \rightarrow_p \mathit{Store} \\ \mathit{Proc}_r & = & (\mathit{Store} \times \mathit{Loc}) \rightarrow_p \mathit{Store} \\ \mathit{Proc} & = & \mathit{Proc}_0 + \mathit{Proc}_v + \mathit{Proc}_r \\ \mathit{Env} & = & \mathcal{X} \rightarrow_{\mathit{fin}} (\mathit{Loc} + \mathit{Proc}) \end{array}$$

The corresponding closures used in the operational world now carry the formal parameters (marked with the name of the parameter-passing mechanism) in addition to the body of the procedure and its definition environment.

The operational rules for the new constructs are given below. In the rule $Call_v$, γ_1 is the environment of the declaration of the procedure P. It is necessary to allocate a new location l for the formal parameter x in which the value of the actual parameter obtained by evaluating e in the state $\langle \gamma,\ \sigma \rangle$ is stored. Finally, of course, the location l needs to be made inaccessible on exit from the procedure. Hence the conclusion of the rule has the restriction of σ' to the domain of σ. The presence of the binding for P in γ_2 is a simple expedient to deal with recursion.

$$Sub_v \quad \frac{}{\langle \gamma,\ \sigma,\ \textbf{sub}\ P\ (\textbf{val}\ x)\ =\ c \rangle \Longrightarrow_d \langle [P \mapsto \texttt{proc}\langle \textbf{val}\ x, c, \gamma \rangle],\ \sigma \rangle}$$

$$Call_v \quad \frac{\gamma, \sigma\ \vdash\ e\ \Longrightarrow_e\ v \qquad \langle \gamma_2,\ \sigma[l \mapsto v],\ c \rangle \Longrightarrow_c \sigma'}{\langle \gamma,\ \sigma,\ \textbf{call}\ P(e) \rangle \Longrightarrow_c \sigma' \wr\ dom(\sigma)}$$

where $\gamma_2 = \gamma_1[P \mapsto \gamma(P)][x \mapsto l]$
$l \notin codom(\gamma) \cup dom(\sigma)$
and $\gamma(P) = \texttt{proc}\langle \textbf{val}\ x, c, \gamma_1 \rangle$.

In a similar vein we also define the semantics of procedures that use a reference parameter. Note that the formal parameter x is now associated with the location of the actual parameter y in invocation **call** $P(y)$. There are no new locations created, hence $dom(\sigma) = dom(\sigma')$. The effect of updating the formal parameter x within the procedure body, is directly reflected in the contents of the location of the actual parameter.

$$Sub_r \quad \frac{}{\langle \gamma,\ \sigma,\ \textbf{sub}\ P\ (\textbf{var}\ x)\ =\ c \rangle \Longrightarrow_d \langle [P \mapsto \texttt{proc}\langle \textbf{var}\ x, c, \gamma \rangle],\ \sigma \rangle}$$

$$Call_r \quad \frac{\langle \gamma_2,\ \sigma,\ c \rangle \Longrightarrow_c \sigma'}{\langle \gamma,\ \sigma,\ \textbf{call}\ P(y) \rangle \Longrightarrow_c \sigma'}$$
where $\gamma_2 = \gamma_1[P \mapsto \gamma(P)][x \mapsto \gamma(y)]$

3.4 Run-time Allocation and Deallocation

One of the most nettlesome features of most programming languages is the use of pointers – their creation, access and disposal. Pointers are a major

source of problems for users, implementors and language designers alike. It is therefore necessary to precisely define the semantics of dynamic memory allocation and deallocation. This is also a feature easier to treat operationally rather than denotationally.

Briefly, one of the problems with pointers is *aliasing*. The problem of aliasing is not an exceptional circumstance, since it is often the case that distinct dereferencing expressions refer to the same location on the heap. Hence an assignment to one of the references might alter the value of some other seemingly unrelated expression. The second major problem is that it is fairly common to work with several logically distinct data structures in heap, where there is sharing of components. Thirdly, while discussing memory allocation and deallocation, it is important to treat definedness (a major source of run-time errors).

Recently, [CIO00] have built on some previous work of Morris [Mor82] and Bornat [Bor00] to specify the semantics of aliasing, memory allocation and disposal. For simplicity, the store is assumed to consist of two components – a stack, which holds the values of local variables, and a heap which contains data that is dynamically created and destroyed. Naturally any access to the heap is from the stack. Any structure that is inaccessible from the stack is treated as garbage. The stack can be extended by declarations of local variables and variable values can be modified by assignments. The heap on the other hand, is assumed to consist of only one kind of data structure, namely, records, where each record has a fixed number of components indexed by tags.

We extend the language of expressions to include record component access and update. The meta-variables $a, b, \ldots$ range over tags (record components). An expression can also be a null pointer or access to a record component.

$$e \quad ::= \quad \ldots \mid e.a$$

Correspondingly, the domain of denotable values for expressions is extended to include locations and a special value $null$. The changes that are needed in the various domain definitions are listed below:

$$\begin{array}{lcl} \mathit{Tag} & = & \{a, b, \ldots\} \\ \mathcal{V} & = & \ldots + \mathit{Loc} + \{\mathit{null}\} \\ \mathit{Stack} & = & \mathcal{X} \rightarrow_{\mathit{fin}} \mathcal{V} \\ \mathit{Heap} & = & \mathit{Loc} \rightarrow_{\mathit{fin}} (\mathit{Tags} \rightarrow \mathcal{V}) \\ \mathit{Store} & = & \mathit{Stack} \times \mathit{Heap} \end{array}$$

A store σ is a pair (st, hp), containing a stack st and a heap hp. Both the stack st and the heap hp are partial functions. Their domains are denoted $dom(st)$ and $dom(hp)$ respectively. $dom(st)$ includes only the variables in the current scope and $dom(hp)$ includes only the locations allocated so far and is finite.

Two distinct variables x and y could point to the same record on the heap i.e. $x.a$ and $y.a$ could be aliases. However, two distinct variables cannot be aliases since variables are on stack and not on the heap. Moreover the "l-values" of variables cannot be modified. For any variable x which may be a pointer to a record on the heap, $x.a$ represents access to a component a.

Example 3.6 *We restrict ourselves to the two constructors for linked lists — hd and tl respectively. For any list variable x (on stack) $x.hd$ will denote the "value" of the first element in the list (if the list is nonempty), whereas $x.tl$ will denote a location from which the rest of the list is accessible. Hence $x.tl.hd$ will be the value of the second element of the list (if one does exist). We also allow for a special value null to be stored in x (to denote the empty list). Hence if the list (ML-style)* `[1, 2, 3]` *is the value of the variable x on stack, then we require $x \in dom(st)$ and three locations $\{l_1, l_2, l_3\} \subseteq dom(hp)$ such that*

$$\begin{array}{llllllll} st(x) & = & l_1 & & & & & \\ hp(l_1)(hd) & = & 1 & , & hp(l_1)(tl) & = & l_2 \\ hp(l_2)(hd) & = & 2 & , & hp(l_2)(tl) & = & l_3 \\ hp(l_3)(hd) & = & 3 & , & hp(l_3)(tl) & = & null \end{array}$$

Clearly it follows that $x.tl.tl.hd = 3$ where "." is left associative.

The operational rule for the new expression is given below. The rules for other expressions are as given in Table 4. We use the meta-variable l to range over $Loc + \{null\}$, and v will range over "actual values" that are not locations.

$$ref_{loc} \quad \frac{(st, hp) \vdash e \Longrightarrow_e l \qquad l \in dom(hp)}{(st, hp) \vdash e.a \Longrightarrow_e hp(l)(a)}$$

Since it is now possible for assignment commands to allow the assignments of pointer expressions, we require two rules for the assignment command. The first rule defines the assignment of values to variables on the stack.

Depending upon the type of the variable, it may either be an integer value or a location[5]. We use the meta-variable vl to denote that it may be drawn from either values or $Loc + \{null\}$.

We use $h[l.a \mapsto v]$ to abbreviate $h[l \mapsto (h(l)[a \mapsto v]]$. The rules for assignment are shown below.

$$assign_{var} \quad \frac{(st, hp) \vdash e \Longrightarrow_e vl}{\langle (st, hp), \quad x := e \rangle \Longrightarrow_c (st[x \mapsto vl], hp)}$$

$$assign_{ref} \quad \frac{(st, hp) \vdash e_1 \Longrightarrow_e l \quad (st, hp) \vdash e_2 \Longrightarrow_e vl \qquad l \in dom(hp)}{\langle (st, hp), \quad e_1.a := e_2 \rangle \Longrightarrow_c (st, hp[l.a \mapsto vl])}$$

Having defined the semantics of references, we are now ready to augment the language with commands for allocation and deallocation of memory. We then extend the language of commands to include the two Pascal-like commands.

$$c \quad ::= \quad \ldots \mid \mathbf{new}(x) \mid \mathbf{free}(e)$$

$\mathbf{new}(x)$ will non-deterministically select a location not currently in $dom(hp)$ and initialize the record with the value "$\perp$". We use $hp[l.* \mapsto \perp *]$ to denote that all components of the record $hp(l)$ are initialized to $\perp$. Similarly, $\mathbf{free}(e)$ simply removes the location denoted by e from $dom(h)$. The rules are given below.

$$new \quad \frac{l \notin dom(hp)}{\langle (st, hp), \ \mathbf{new}(x) \rangle \implies (st[x \mapsto l], hp[l.* \mapsto \perp *])}$$

$$free \quad \frac{(st, hp) \vdash e \Longrightarrow_e l \qquad l \in dom(hp)}{\langle (st, hp), \ \mathbf{free}(e) \rangle \implies (st, hp - l)}$$

In the rule for $\mathbf{free}(e)$, $h - l$ denotes the fact that the heap is no longer defined for l (as opposed to being filled with value "$\perp$").

The above rules give us a flavor of how operational rules may be used to specify implementation intuition to a large extent. In [CIO00], the authors also show how these rules may be used to justify axiomatic rules for reasoning locally about aliasing and dynamic memory allocation and deallocation.

[5]The issue of types is something that needs to be addressed by a static semantics, as pointed out elsewhere. It is not properly the domain of a dynamic semantics. So we will continue to believe that all the constructs we use are type-safe.

4 Functions and higher-order forms

Applying the *principle of abstraction* [Ten81] to expressions or commands allows us to form *abstracts* that may be *invoked*, usually with different *parameters*. These abstract forms are called *functions* and *procedures* respectively. Abstract expressions (with a single parameter) are written as $\lambda x.e$. λ *binds* the variable x within the scope of the "body" e. An abstract a can be invoked by "applying" it to an argument e, written as $(a\ e)$; such calls belong to the syntactic category over which the abstract is formed.

The situation becomes more interesting in "higher-order languages" which admit such abstracts as *first-class* values – abstracts can themselves be bound to variables, passed as arguments and returned as results of other functions. The various issues related to functions and procedures, in particular the correct formulation of lexical scoping and of recursive function definitions, can be explored in a higher-order functional language with only single parameter functions (the generalizations to procedures and multiple parameters is a matter of detailing, but does not need very much by way of new concepts). Indeed, these two issues are of vital importance — early implementations of Lisp implemented "dynamic scoping" because of a rather simplistic implementation of recursion.

Exp is now extended to

$$e \ ::= \ \ldots \lambda x.e \mid (e_1\ e_2)$$

We look at an extremely simple quintessential functional language, called the λ-calculus. Indeed, Landin explicated the block structured features of Algol by relating them to the λ-calculus. The operational semantics for the λ-calculus is given in a *purely syntactic manner* (involving no extra-syntactic constructs such as environments). From these, various environment-based formulations can be constructed to realize the semantics in an efficient manner.

4.1 λ-calculus

The syntax of the "pure" λ-calculus is:

$$e \ ::= \ x \mid \lambda x.e_1 \mid (e_1\ e_2)$$

Expressions (or terms) are variables, *abstractions* on expressions, or *applications* of one expression (putatively a function) to another (an argument).

Other kinds of values and expressions such as those we have examined so far can be added together with their computation rules to obtain an *applied λ-calculus*. While applied λ-calculi raise interesting issues and problems, the pure calculus itself exhibits several important concepts. Plotkin's seminal papers [Plo75, Plo77] are good examples of detailed studies of many of the fundamental issues.

Definition 4.1 (free and bound variables) *An occurrence of a variable x in a term e is bound if it appears in a sub-term $\lambda x.e'$. All occurrences of variables that are not bound or binding are* free. *The function fv returns the set of free variables in a term.*

$$fv(x) = \{x\} \quad fv(\lambda x.e) = fv(e) - \{x\} \quad fv((e_1\ e_2)) = fv(e_1) \bigcup fv(e_2)$$

Bound variables may be systematically renamed without altering the intended meaning of an expression. By systematic, we mean that two hitherto different variables are not suddenly identified, in particular that no previous free variable is suddenly "captured" and bound. We identify expressions that differ only in the choice of names of bound variables, a notion called α-equivalence. Expressions with no free variables are called closed.

The major meta-operation for syntactic manipulation in any λ-calculus is *substitution*.

Definition 4.2 (substitution) *We write $e[e'/x]$ to denote the term obtained by substituting e' for all free occurrences of variable x in term e. Substitution is defined using a case analysis on e*[6]*:*

$$x[e'/x] = e'$$

$$y[e'/x] = y \qquad\qquad y \not\equiv x$$

$$(e_1\ e_2)[e'/x] = (e_1[e'/x]\ e_2[e'/x])$$

$$\lambda y.e_1[e'/x] = \lambda z.(e_1[z/y][e'/x]) \qquad z \notin fv(e_1) \bigcup fv(e')$$

[6]This version of the definition "factors in" α-equivalence whereas most treatments do not.

$$(\beta) \quad \frac{}{(\lambda x.e_1\ e_2)\ \longrightarrow_1\ e_1[e_2/x]}$$

$$(\xi) \quad \frac{e\ \longrightarrow_1\ e'}{\lambda x.e\ \longrightarrow_1\ \lambda x.e'}$$

$$(op) \quad \frac{e_1\ \longrightarrow_1\ e_1'}{(e_1\ e_2)\ \longrightarrow_1\ (e_1'\ e_2)}$$

$$(arg) \quad \frac{e_2\ \longrightarrow_1\ e_2'}{(e_1\ e_2)\ \longrightarrow_1\ (e_1\ e_2')}$$

Table 6: β-reduction in the λ-calculus

Since substitution avoids capture of free names, it perforce avoids the possibility of accidental dynamic binding.

It is often convenient to use the notion of *contexts* in examining the structure of a term.

Definition 4.3 (context) *A context is a λ-term with a "hole" given by the following abstract grammar:*

$$C\ ::=\ [\ ]\ |\ (C\ C)\ |\ \lambda x.C\ |\ e$$

One-hole contexts are characterized as

$$C^1\ ::=\ [\ ]\ |\ (C^1\ e)\ |\ (e\ C^1)\ |\ \lambda x.C^1$$

Definition 4.4 (reduction) *A redex is any term of the form $(\lambda x.e_1\ e_2)$. Any term containing a redex as a sub-term is called reducible. The β-reduction rule is*

$$C^1[((\lambda x.e_1)\ e_2)]\ \longrightarrow_1\ C^1[e_1[e_2/x]]$$

where $C^1[\ ]$ is any one-hole context.

An alternative formulation of β-reduction is given in Table 6.

Some important results about β-reduction are:

Lemma 4.5 (Substitution and β-reduction) *If* $e \longrightarrow_1 e'$ *then* $e_1[e/x] \ (\longrightarrow_1)^* \ e_1[e'/x]$ *and* $e[e_1/x] \longrightarrow_1 e'[e_1/x]$.

Proposition 4.6 (Local confluence) *β-reduction satisfies the weak diamond property.*

Theorem 4.7 (Church-Rosser) *β-reduction is confluent.*

Theorem 4.8 (Standardization) *If* $e(\longrightarrow_1)^* e'$ *then* $e(\longrightarrow_1^{standard})^* e'$ *by always reducing the leftmost outermost redex at each stage.*

Proposition 4.9 (fixed points) *There exists a closed λ-calculus term Y, called a fixed point combinator, such that* $(Y \ e) \longrightarrow_1^* (e \ (Y \ e))$, *for any* e.

4.2 Relationship with functional languages.

Almost all functional languages disallow reduction "below" a λ — redexes appearing in terms of the form $\lambda x.e$ are not considered. In other words, such "weak reduction" does not have the ξ-rule. Hence, not all results (confluence!) shown for the λ-calculus automatically transfer to functional languages based on them. Moreover, certain results do not hold for typed frameworks. For instance, fixed point combinators do not exist in simply typed lambda calculi[7]. Finally, we should mention that programming languages are concerned with closed terms only.

Two commonly used strategies for reducing terms are (weak) *call-by-value* (or eager) and (weak) call-by-name (or lazy)[8]. In what follows, we present different formulations of these two strategies and how they are realized.

Call-by-value. The basic notion in call-by-value (*cbv*) is that arguments to a function are evaluated before evaluation of the function body commences. The notion of *value* is crucial — they are merely all abstractions: $v \in Val \ ::= \lambda x.e$. Values are irreducible, but not conversely.

We first present the big-step formulation for call-by-value reduction:

$$\frac{}{v \Longrightarrow_v v} \qquad \frac{e_1 \Longrightarrow_v \lambda x.e_1' \quad e_2 \Longrightarrow_v v_2 \quad e_1'[v_2/x] \Longrightarrow_v v}{(e_1 \ e_2) \Longrightarrow_v v}$$

In the small-step framework, this is formulated as shown in Table 7.

[7] though they can in languages with reflexive types or recursive types

[8] Various researchers actually distinguish between call-by-value and eager (or call-by-name and lazy) which we gloss over here.

$$(\beta_v) \quad \frac{}{(\lambda x.e_1\ v) \ \longrightarrow_1^v \ e_1[v/x]}$$

$$(op) \quad \frac{e_1 \ \longrightarrow_1^v \ e_1'}{(e_1\ e_2) \ \longrightarrow_1^v \ (e_1'\ e_2)}$$

$$(arg_v) \quad \frac{e_2 \ \longrightarrow_1^v \ e_2'}{(v\ e_2) \ \longrightarrow_1^v \ (v\ e_2')}$$

Table 7: Call-by-value β-reduction

Alternatively, the cbv strategy can be explained by using the β_v reduction rule in the following *cbv evaluation contexts*:

$$E_v^1 \ ::= \ [\ ] \mid (E_v^1\ e) \mid (v\ E_v^1)$$

Call-by-name. Call-by-name (*cbn*), in contrast, does not simplify arguments before function call. The big-step *cbn* rules are:

$$\frac{}{v \Longrightarrow_n v} \qquad \frac{e_1 \Longrightarrow_n \lambda x.e_1' \quad e_1'[e_2/x] \Longrightarrow_n v}{(e_1\ e_2) \Longrightarrow_n v}$$

Note that arguments are not evaluated before substituting them for the formal parameter in the function body. This may result in more than one copy of the same argument, which may be evaluated multiple times. The advantage of *cbn* over *cbv* is that arguments that are not needed are not evaluated. An important static analysis technique is *strictness analysis*, in which *cbv* evaluation can safely be used instead of *cbn*. An alternative formulation of the *cbn* rules is given in Table 8.

Alternatively, the *cbn* strategy can be explained by using the β rule in the following *cbn evaluation contexts*:

$$E_n^1 \ ::= \ [\ ] \mid (E_n^1\ e)$$

Context machines. The notion of evaluation context permits a simple transformation, due to Felleisen and Wright [WF94] of reduction semantics into an abstract machine. We illustrate the idea for *cbv* reduction. A similar machine can be constructed for *cbn* reduction.

$$(\beta) \quad \frac{}{(\lambda x.e_1\ e_2)\ \longrightarrow_1^n\ e_1[e_2/x]}$$

$$(op) \quad \frac{e_1\ \longrightarrow_1^n\ e_1'}{(e_1\ e_2)\ \longrightarrow_1^n\ (e_1'\ e_2)}$$

Table 8: Call-by-name β-reduction

We first characterize *basic evaluation contexts* F^v:

$$F^v\ ::=\ ([\ ]\ e)\ |\ (\lambda x.e\ [\ ])$$

Using the fact that any non-trivial *cbv* evaluation context can be expressed as the composition of basic evaluation contexts $F_1^v[F_2^v[...F_k^v[\ ]...]]$, (the trivial context $[\ ]$ can be considered as corresponding to the case where $k = 0$), we define a "context stack machine" as follows. The machine has two components — a stack of basic evaluation contexts FS, and the current expression e. Transitions are defined by cases depending on the structure of e and then of FS:

$$\langle \left\lfloor \begin{array}{c} ([\ ]\ e) \\ FS \end{array} \right\rfloor,\ v\rangle \longrightarrow \langle \left\lfloor \begin{array}{c} (v\ [\ ]) \\ FS \end{array} \right\rfloor,\ e\rangle$$

$$\langle \left\lfloor \begin{array}{c} ((\lambda x.e)\ [\ ]) \\ FS \end{array} \right\rfloor,\ v\rangle \longrightarrow \langle \left\lfloor\ FS\ \right\rfloor,\ e[v/x]\rangle$$

$$\langle \left\lfloor\ FS\ \right\rfloor,\ (e_1\ e_2)\rangle \longrightarrow \langle \left\lfloor \begin{array}{c} ([\ ]\ e_2) \\ FS \end{array} \right\rfloor,\ e_1\rangle$$

The machine is started in configuration $\langle \left\lfloor\ \right\rfloor,\ e\rangle$ for any closed e and terminates with context stack empty and value v.

Now if we define function *crunch* as:

$$crunch\langle \left\lfloor\ \right\rfloor,\ e\rangle \quad = \quad e$$

$$crunch\langle \left\lfloor \begin{array}{c} F_n^v \\ FS \end{array} \right\rfloor,\ e\rangle \quad = \quad crunch\langle \left\lfloor\ FS\ \right\rfloor,\ F_n^v[e]\rangle$$

it is easy to show that

$$\langle \lfloor\ FS\ \rfloor,\ e\rangle \ \longrightarrow\!\triangleright^*\ \langle \lfloor\ \ \rfloor,\ v\rangle \text{ if and only if } \mathit{crunch}\langle \lfloor\ FS\ \rfloor,\ e\rangle \ \Longrightarrow_v\ v$$

4.3 Closures and Environment machines

As mentioned earlier in passing, substitution is an expensive operation, since it involves traversing the term in which the substitution is being performed (as well as α-conversion to prevent name capture). Environments are a convenient ancillary structure used to record the bindings for variables in substitutions.

Closures. Suppose environments were, as before, represented by finite domain functions from variables to "values". Suppose we considered an environment γ in which f was bound to $\lambda x.e$ and proposed a rule for function call:

$$\frac{\gamma\ \vdash\ e_1 \Longrightarrow_e v_1 \qquad \gamma[x \mapsto v_1]\ \vdash\ e \Longrightarrow_e v}{\gamma\ \vdash\ f(e_1) \Longrightarrow_e v}$$

The problem with this rule is that if e contains free variables other than x, lexical scoping may be violated if the binding for f was made in an environment other than γ, since in the call, they will take their value (if they can) from γ. While the problem is more acute in higher-order languages, it nevertheless exists in simple block structured procedures as well, which is why we disallowed nested procedures in §3.3. It is therefore necessary to "package" in when making the binding for f the prevalent environment. Such a pair is called a *closure*. We define

$$\mathit{Clos} \subseteq \mathit{Exp} \times \mathit{Env} \qquad \mathit{Env} = \mathcal{X} \rightarrow_{\mathit{fin}} \mathit{Clos}$$

In an applied calculus, there can be other kinds of values apart from closures.

Closures permit a correct treatment of lexical scope, and thus remedy the lacuna in our treatment of procedures. They can also correctly handle recursive functions (and other recursive data structures that are possible in a lazy language). Let *vcl* range over closures of the form $\ll \lambda x.e, \gamma \gg$. We give a big-step description for *cbn* and *cbv* simplifications of closures, which are basically restatements of the rules for $\Longrightarrow_n$ and $\Longrightarrow_v$. Very roughly, the judgments used for closure evaluation under strategy X $\ll e, \gamma \gg\ \Longrightarrow_{cl}^{X}\ \mathit{vcl}$

correspond to judgments $\gamma \vdash e \implies_X v$ for expression evaluation, and where value closure vcl "unravels" to value v.

$$\frac{\gamma(x) \implies^n_{cl} vcl}{\ll x, \gamma \gg \implies^n_{cl} vcl}$$

$$\frac{\ll e_1, \gamma \gg \implies^n_{cl} \ll \lambda x.e', \gamma' \gg \qquad \ll e', \gamma'[x \mapsto \ll e_2, \gamma \gg] \gg \implies^n_{vcl} cl}{\ll (e_1\ e_2), \gamma \gg \implies^n_{cl} vcl}$$

For cbv the rules are:

$$\frac{\gamma(x) \implies^v_{cl} vcl}{\ll x, \gamma \gg \implies^v_{cl} vcl}$$

$$\frac{\ll e_1, \gamma \gg \implies^v_{cl} \ll \lambda x.e', \gamma' \gg \qquad \ll e_2, \gamma \gg \implies^v_{cl} vcl_2 \qquad \ll e', \gamma'[x \mapsto vcl_2] \gg \implies^n_{cl} vcl}{\ll (e_1\ e_2), \gamma \gg \implies^n_{cl} vcl}$$

It is also possible to formulate a calculus of closures [Cur91] and study properties such as confluence of its reduction relation, which is "weak" in the sense that reduction does not occur below abstractions.

Abstract machines. The big-step semantics suggests using a stack of closures that are yet to be simplified, or which are awaiting their arguments. Using this insight, environment machines can be developed, manipulating closures.

An environment machine for cbn due to Krivine is:

$$\langle \ll x, \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \gamma(x), \lfloor\ S\ \rfloor \rangle$$

$$\langle \ll (e_1\ e_2), \gamma \gg, S \rangle \longrightarrow \langle \ll e_1, \gamma \gg, \left\lfloor \begin{array}{c} \ll e_2, \gamma \gg \\ S \end{array} \right\rfloor \rangle$$

$$\langle \ll \lambda x.e, \gamma \gg, \left\lfloor \begin{array}{c} cl \\ S \end{array} \right\rfloor \rangle \longrightarrow \langle \ll e, \gamma[x \mapsto cl] \gg, \lfloor\ S\ \rfloor \rangle$$

The machine configurations consist of a *current* closure to be simplified and a stack of closures which are (yet-to-be evaluated) arguments to the current term. The first rule is a look-up. The second rule stacks the closure consisting

of argument N together with the current environment (in which it should be evaluated) onto the stack of yet-to-be-evaluated closures. The third rule starts the evaluation of the body in a closure after extending the environment with a binding of formal x to the argument closure, which is atop the stack.

The corresponding environment machine for *cbv* is:

$$\langle \ll x, \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \gamma(x), \lfloor\ S\ \rfloor \rangle$$

$$\langle \ll (e_1\ e_2), \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \ll e_1, \gamma \gg, \left\lfloor \begin{array}{c} \searrow \\ \ll e_2, \gamma \gg \\ S \end{array} \right\rfloor \rangle$$

$$\langle vcl, \left\lfloor \begin{array}{c} \searrow \\ \ll e, \gamma \gg \\ S \end{array} \right\rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \left\lfloor \begin{array}{c} \swarrow \\ vcl \\ S \end{array} \right\rfloor \rangle$$

$$\langle vcl, \left\lfloor \begin{array}{c} \swarrow \\ \ll \lambda x.e, \gamma \gg \\ S \end{array} \right\rfloor \rangle \longrightarrow \langle \ll e, \gamma[x \mapsto vcl] \gg,\ S \rangle$$

The *cbv* machine is not much different, except that both operator and operand are to be evaluated before application. For this, two markers $\searrow$ and $\swarrow$ are used to indicate that the closure below it on the stack is the argument and operator respectively of an application. The third rule swaps the evaluated operand and unevaluated operators between the current-closure and the top-of-stack positions.

Both machines are loaded with a closure consisting of a closed term and empty environment, with an empty stack. The *unload* function involves unfolding the resulting closure, using the packaged environment to obtain the terms bound to variables (recursively unfolding closures).

SECD Machine. The prototypical machine used for *cbv* evaluation of a functional language was the *SECD* machine [Lan65a]. This machine works with two stacks — S for already evaluated expressions and "dump" D for managing control during function call and return — an environment E and a list of opcodes C. Stack S is used in much the same way as the stack is used for expression evaluation — the closures to which expressions evaluate are pushed onto it. Dump D is used as a repository for storing the calling context

(the current environment, the sub-expressions already evaluated prior to the call, and the code to be evaluated after the call) when a function call is made; this context can then be restored from the top of the dump on completion of a function call. To avoid introducing new symbols, we use (following [Plo75]) the λ-terms themselves as op-codes, with one additional op-code for function application *app*.

$$\langle \left\lfloor \begin{array}{c} cl \\ S \end{array} \right\rfloor, \gamma, \epsilon, \left\lfloor \begin{array}{c} \langle S',\ \gamma',\ c' \rangle \\ D \end{array} \right\rfloor \rangle \longrightarrow \langle \left\lfloor \begin{array}{c} cl \\ S' \end{array} \right\rfloor, \gamma', c', D \rangle$$

$$\langle \left\lfloor\ S\ \right\rfloor, \gamma, x :: c, \left\lfloor\ D\ \right\rfloor \rangle \longrightarrow \langle \left\lfloor \begin{array}{c} \gamma(x) \\ S \end{array} \right\rfloor, \gamma, c, \left\lfloor\ D\ \right\rfloor \rangle$$

$$\langle \left\lfloor\ S\ \right\rfloor, \gamma, \lambda x.e :: c, \left\lfloor\ D\ \right\rfloor \rangle \longrightarrow \langle \left\lfloor \begin{array}{c} \ll \lambda x.e, \gamma \gg \\ S \end{array} \right\rfloor, \gamma, c, \left\lfloor\ D\ \right\rfloor \rangle$$

$$\langle \left\lfloor\ S\ \right\rfloor, \gamma, (e_1\ e_2) :: c, \left\lfloor\ D\ \right\rfloor \rangle \longrightarrow \langle \left\lfloor\ S\ \right\rfloor, \gamma, e_1 :: e_2 :: app :: c, \left\lfloor\ D\ \right\rfloor \rangle$$

$$\langle \left\lfloor \begin{array}{c} cl \\ \ll \lambda x.e, \gamma' \gg \\ S \end{array} \right\rfloor, \gamma, app :: c, \left\lfloor\ D\ \right\rfloor \rangle \longrightarrow \langle \lfloor\rfloor, \gamma'[x \mapsto cl], e, \left\lfloor \begin{array}{c} \langle S,\ \gamma,\ c \rangle \\ D \end{array} \right\rfloor \rangle$$

The first rule describes function return; it says that if the current call has no remaining instructions, the calling context is restored from the dump — the returned value placed atop the caller's stack, and the environment and code list of the caller are restored. The second rule is a variable look up. The third rule forms and places a closure atop the value stack. The fourth rule is really a "compilation rule" which evaluates operator and operand expressions of an application (it is possible to separate the execution and compilation phases). The fifth rule is the actual function call rule. It assumes that the operand (argument) closure *cl* sits above the operator (function) closure atop the stack. *cl* is bound to the formal argument x in the operator closure's environment, the operator closure's code is now made the code list, and the calling context is placed atop the dump. As indicated above, the calling context consists of the stack below the operator and operand closures, the calling environment and the remaining code list.

The *SECD* machine has been used as a template for a variety of block-structured languages, as we will discuss below. Plotkin [Plo75] has related the

abstract machine semantics with the big-step and reduction semantics of an applied *cbv* λ-calculus using standardization to establish the correspondence.

Other abstract machines. There are various other abstract machine implementations that we cannot describe here. One such class of machines is based on a translation of the λ-calculus into combinatory logic and an implementation of these combinators [Tur79]. A special class of implementations are based on *graph reduction* (see [Jon87] and various references therein for an accessible treatment of such implementations). The main operations of these machines involve performing rearrangements of a syntax tree (or graph) according to certain combinators or directors [KS88]. Also significant is the Categorical Abstract Machine [CCM85] which is based on operative features of categorical models of λ-calculi, and the closely related machine derived by Hannan and Miller [MH90].

4.4 Implementation issues related to environments

The abstract machines seem rather profligate in the structures they employ. Fortunately, there are rather efficient implementations of environments, and closures using stacks, pointers and allocation on stack and heap. The observation that the called function never looks at the caller's stack in the *SECD* machine suggests that the value stack does not need storing, only the (re)storing of the stack pointer. Likewise, entire code lists and environments need not be stowed away on the dump, pointers to them will suffice.

Efficient environment implementation and management is crucial. First, the environment is maintained as a stack of references to local *frames*. Then, variables are replaced by a fast indexing scheme relative to a frame pointer (*c.f.* de Bruijn indices in the λ-calculus).

Recursion. Special mention must be made about recursive functions. As mentioned above, simply typed languages cannot have a Y combinator, so a special mechanism is needed to build closures for recursive functions and recursive data structures. A simple idea is to build a circular reference into the environment component of the closure for a recursive function. This is achieved using two op-codes introducing a level of indirection in environments[9]. The first op code places a *reference* to a dummy closure. The closure

[9] which is already there in most pointer-based implementations of environments

for the recursive function is created using this augmented environment, and a second op-code overwrites the reference to the dummy reference with a pointer to the new closure, thus building the cycle (see [Hen80, Car84] for a simple implementation).

Local definitions. Local definitions may be implemented in correspondence to the parameter-passing mechanism, employing the equivalence

$$(\lambda x.e_2\ e_1)\ \approx\ \textbf{let}\ x \stackrel{def}{=} e_1\ \textbf{in}\ e_2$$

or its generalization to more structured definitions. However such a crude approach is rarely followed, since it is inefficient. Exploiting the fact that the environment used for e_2 is an extension of that used for e_1, much simpler and direct methods are possible, in particular, by employing finer grain op-codes that facilitate stack manipulation and making definitions and recursive definitions.

Extensions. The *SECD* framework is fairly robust, and can easily be extended to deal with a variety of language extensions, including side effects. Adding a store component and related op-codes [Car86a] allows support for imperative features. Similarly, input and output streams can be accommodated, as also can communication and concurrency primitives (a general choice operator is difficult to incorporate) [GMP89].

Procedures in imperative languages. By the principle of abstraction, the notion of closures carries over to command abstracts. Of course, there are some aspects that are simpler (languages with higher-order procedures are rare beasts), whereas issues pertaining to stores are somewhat more involved. In particular, showing that the allocation and deallocation of locations is done correctly is an important part of proving that the language and implementation are free of storage insecurities.

The typical call-stack management in traditional imperative languages can be seen as an implementation where three different stack structures – S for temporary computation, E for the environment and D for the dump — are "multiplexed" onto one physical stack.

4.5 Control operators

We briefly discuss here the operational semantics for an extension of the λ-calculus with *control operators* that can pass or throw away the current evaluation context. Control operators allow functional programs to handle features like concurrent threads, exceptions, *call/cc* etc. They support a technique used in modern compilers, namely that of passing *continuations* [App92]. Moreover, the environment machines given earlier have simple extensions to deal with these new control operators.

The syntax is extended with two new unary operations, which are also redexes:

$$e \ ::= \ \ldots \ \mid \ \mathcal{C}e \ \mid \ \mathcal{A}e$$

whose reduction rules, stated in contextual form, are:

$$(\mathcal{C}) \qquad \frac{}{E[\mathcal{C}e] \ \longrightarrow_1^e \ (e \ (\lambda x.\mathcal{A}E[x]))} \qquad x \notin fv(e)$$

$$(\mathcal{A}) \qquad \frac{}{E[\mathcal{A}e] \ \longrightarrow_1^e \ e}$$

In the rule $(\mathcal{A})$, the "abort" operator throws away the current evaluation context, whereas in the rule $(\mathcal{C})$, the "control" operator passes an abstracted form of the current evaluation as an argument to the expression e.

Another well-known control operator is *call/cc*, "call with current continuation", with the following operational rule:

$$(call/cc) \qquad \frac{}{E[call/cc(\lambda k.e)] \ \longrightarrow_1^e \ ((\lambda k.(k \ e)) \ (\lambda x.\mathcal{A}E[x]))} \qquad x \notin fv(e)$$

an equivalent of which can be expressed in terms of $(\mathcal{C})$ and $(\mathcal{A})$.

Environment machines for control operations. Recall that the stack component S of an environment machine represents the context E of the current expression being evaluated. The control operators manipulate this evaluation context. Therefore, operations to encapsulate and manipulate the stack are introduced: A new kind of closure $retr(S)$ is added that corresponds roughly to $\lambda x.\mathcal{A}E[x]$.

The *new* rules for the Krivine machine are:

$$\langle \ll \mathcal{C}e, \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \lfloor\ retr(S)\ \rfloor \rangle$$

$$\langle \ll \mathcal{A}e, \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \lfloor\ \ \rfloor \rangle$$

$$\langle retr(S), \left\lfloor \begin{array}{c} cl \\ S' \end{array} \right\rfloor \rangle \longrightarrow \langle cl, \lfloor\ S\ \rfloor \rangle$$

The manipulations of the context are fairly clear: in the first rule, the current stack is encapsulated and presented as an argument to the closure corresponding to e. The "abort" operator throws away the current stack. In the third rule, the encapsulated stack is restored, in place of the existing stack S'.

The *cbv* environment machine uses the same rules as before with three additional rules for manipulating the stack. Of these, the second rule (for abort) is the same as the rule in the extension of the Krivine machine.

$$\langle \ll \mathcal{C}e, \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \left\lfloor \begin{array}{c} \searrow \\ retr(S) \end{array} \right\rfloor \rangle$$

$$\langle \ll \mathcal{A}e, \gamma \gg, \lfloor\ S\ \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \lfloor\ \ \rfloor \rangle$$

$$\langle vcl, \left\lfloor \begin{array}{c} \swarrow \\ retr(S) \\ S' \end{array} \right\rfloor \rangle \longrightarrow \langle vcl, \lfloor\ S\ \rfloor \rangle$$

If $retr(S)$ corresponds to $\lambda x.\mathcal{A}E[x]$, and S' corresponds to context $E'[\ ]$, then the last rule can be seen as implementing the reduction sequence

$$E'[(\lambda x.\mathcal{A}E[x]\ v)] \longrightarrow_1^v E'[\mathcal{A}E[v]] \longrightarrow_1^v E[v].$$

Translating the control operators away. An important result [Plo75, FFKD87, Gri90] is that these control operators can be translated away by so-called "CPS transformations" into purely functional languages. We introduce the idea here only to indicate how operational techniques are used in language translations, since a proper treatment of CPS is well beyond the scope of this chapter. We present one such translation, which lets us interpret call-by-value

reduction as call-by-name reduction [Plo75].

$$\overline{x} = \lambda k.(k\ x) \qquad \overline{(e_1\ e_2)} = \lambda k.(\overline{e_1}\ (\lambda m.(\overline{e_2}\ (\lambda n.((m\ n)\ k)))))$$

$$\overline{a} = \lambda k.(k\ a) \qquad \overline{\mathcal{C}e} = \lambda k.(\overline{e}\ (\lambda m.((m\ (\lambda n.\lambda d.(k\ n)))\ (\lambda x.x))))$$

$$\overline{\lambda x.e} = \lambda k.(k\ (\lambda x.\overline{e})) \quad \overline{\mathcal{A}e} = \lambda k.(\overline{e}\ (\lambda x.x))$$

Various interesting theorems can be shown about this CPS translation. For example:

Theorem 4.10 *For any pure λ-expression e: $(\overline{e}\ (\lambda x.x)) \Longrightarrow_n v$ if and only if $(\overline{e}\ (\lambda x.x)) \Longrightarrow_v v$*

Theorem 4.11 *For any λ-expression e without control operators, and of base type (not of a function type)*[10]*: $e\ (\longrightarrow_1)^*\ v$ if and only if $(\overline{e}\ (\lambda x.x))\ (\longrightarrow_1)^*\ v$.*

5 LTSs and Interactive Programs

The formulations we have presented so far have used transition systems without labels. We have till now concentrated on programs in isolation from their operating environment. However, programs *interact* with their execution environment, at the very least for input and output of data. Even in an isolated computer, there are various interactions with peripheral devices such as disks, printers, file systems and libraries. There are also interactions with forked processes, interrupt handlers etc.

The picture we have so far presented can be sustained when interaction with the environment can be clearly separated from computation. However, programming nowadays is increasingly *interactive*, and all programming languages provide facilities for interaction with the environment. In addition, several languages provide features for concurrent and distributed execution. Interactions can take the form of remote procedure calls, or communication in a network / cluster / distributed computing environment, interspersed in

[10] such expressions can be considered "complete programs" in a typed λ-calculus. The result depends on *strong normalization* of the typed lambda calculus.

the computation. In other words, interaction becomes an integral part of computation.

Central to this kind of interactive computing are the concepts of *process* and *communication* (the texts [Hoa85, Hen88, Mil89] provide excellent introductions to the area). A program and its environment can be considered two processes that communicate with each other. These two processes may themselves consist of collections of interacting processes.

When integrating interaction into computation, certain issues arise in providing structured operational descriptions. Firstly, the Fregean principle of compositionality should still be applicable. Secondly, the fact that processes interact while executing concurrently brings in its own complexity since interactions may alter the state of a program non-deterministically. Thirdly, the fact that a program operates correctly only under circumstances where the environment fulfills certain obligations implies that both the program and its environment (regarded as a process) cooperate in achieving certain goals. Specifications must clearly define interfaces of interaction that constrain the kinds of inputs a process can receive, the outputs it can produce and how it synchronizes with other components in a system. Lastly, one cannot place unreasonable restrictions on the environment. For example, it would be unreasonable to expect that a remote server operate at the same speed as one or several of its clients. Hence concurrent execution in general, implies that different processes execute at different speeds and interactions are the only means of achieving certain synchronizations.

Labels and behavior. Labels are a convenient device to indicate interaction between a program and its environment during execution. They carry information about communication capabilities of processes and are often crucial to the changes in state that processes incur. They are also used to determine and resolve non-deterministic choices in the execution of a process when it has the possibility of interacting with several other processes at the same time.

We saw in TSs that confluence, determinacy and termination were important properties and that two sequential systems are considered equal if they compute the same function between input and output states. Concurrent systems on the other hand, are generally non-deterministic (mostly non-confluent), and often infinite-state, non-terminating systems; neither may they be computing a particular relation or function. So what are the corre-

sponding notions of behavioral properties in LTSs? The crucial properties of such systems concern their interaction capabilities. Any equality relation on such systems will naturally relate to the communication capabilities of the individual processes that make up the system.

Various notions of behavior can be associated with an LTS, based on the idea that the observable behavior of a process depends on the sequences of labelled transitions it can perform. However, there is little consensus yet on what is the right notion of behavior. A simple, language-theoretic notion of program behavior is the set of sequences (finite or infinite) of labels or *traces*. A process p has trace $\varsigma = l_1 l_2 \ldots \in \mathcal{L}^\omega = \mathcal{L}^* \bigcup \mathcal{L}^{\inf}$ if it can perform a sequence of labelled transitions $p \xrightarrow{l_1} p_1 \xrightarrow{l_2} p_2 \ldots$. Two processes are considered *trace-equivalent* if they have the same traces.

Other notions of behavior take into account the communication capabilities (and incapabilities) at each intermediate state, thus being sensitive to the possibility of deadlock – inability to perform a transition with a particular label – in some sequences of transitions (see examples 5.2 and 5.3 below). We present only one such finer notion, called *bisimulation* [Par81].

The intuition is that this notion of equivalence identifies a pair of processes, if starting from equivalent states they have the same interaction possibilities, the success of *each* of which puts them again in states that may be considered equivalent.

Definition 5.1

- *A binary relation $\mathcal{R}$ on process configurations is a* simulation *if whenever $s_1 \mathcal{R} s_2$, for any $l \in \mathcal{L}$, if $s_1 \xrightarrow{l} s_1'$, then there exists a configuration s_2' such that $s_2 \xrightarrow{l} s_2'$ and $s_1' \mathcal{R} s_2'$,*

- *$\mathcal{R}$ is a* bisimulation *if $\mathcal{R}$ and $\mathcal{R}^{-1}$ (the symmetric inverse of $\mathcal{R}$) are both simulations.*

- *The collection of bisimulation relations is closed under inverse, composition and arbitrary union. The largest bisimulation called* bisimilarity *is denoted $\approx$ and is an equivalence relation.*

Proving two labelled transitions systems are bisimilar involves proposing and proving a particular relation is a bisimulation. Bisimulation equivalence or bisimilarity is a finer notion of equivalence than trace equivalence, since it distinguishes more programs than trace equivalence does. In particular, it is sensitive to the potential for deadlock behavior — two processes with the

same traces are distinguished if on some trace, one of them can reach a state where some particular actions are possible whereas the other cannot reach such a corresponding state on that same trace. In fact, bisimilarity is the finest deadlock-sensitive equivalence relation on processes obtained from examining their observable behavior. In practice, there are a variety of notions that can be considered bisimulations, either for different notions of labelled transition, or which differ in the precise characterization of the labelled actions, what exactly is observable, etc. There also may be different characterizations for a single notion of bisimulation, with alternative characterizations supporting different styles of reasoning. There are also a variety of different notions of equivalence that lie between trace equivalence and bisimulation, some of which are fairly natural notions of equivalence to work with. A full exploration of these issues is beyond the scope of this chapter; a quick introduction is provided in [AFV00].

5.1 CSP

We illustrate the use of LTSs in semantic specification through a language based on CSP (*C*ommunicating *S*equential *P*rocesses) due to Hoare [Hoa78, Hoa85]. The language extends the language of guarded choice (which already includes non-determinism) with new constructs for communication and concurrent execution. The semantics we give here is a simplification of a presentation due to Plotkin [Plo83].

We must mention here that it is often difficult to present purely big-step or purely small-step semantics for interactive programming languages, which incorporate internal evaluation of expressions. This is because communicating concurrent systems are best described using small-step descriptions, since they can account for interleavings and interactions from intermediate states (particularly important in notions of behavior sensitive to deadlock), whereas expressions are evaluated in entirety (and can easily be specified in a big-step).

The syntax for CSP is as follows:

$$
\begin{array}{lcl}
io & ::= & P?in \mid Q!out \\
g & ::= & e \mid e; io \\
c & ::= & x := e \mid P?in \mid Q!out \mid c; c \\
 & & \mid \textbf{if } \Box_{i=1}^{n} g_i \triangleright c_i \textbf{ fi} \mid \textbf{do } \Box_{i=1}^{n} g_i \triangleright c_i \textbf{ od} \\
S & ::= & [\ \|_{i=1}^{n} P_i :: c_i\ ]
\end{array}
$$

io stands for input/output communication statements, g for "guards", which are boolean expressions, optionally followed by a communication. Commands are communication statements, assignments, and the guarded choice and iteration constructs. A program S consists of a collection of named processes. For simplicity we assume that concurrent execution takes place only at the topmost level, *i.e.*, processes cannot have subprocesses that themselves execute concurrently. Every process has a name that is known to other processes. Communication between processes is by *synchronized handshaking* or *rendezvous*, wherein two named processes that need to exchange values wait at matching input and output commands respectively before consummating the communication. The command $P?in$ denotes that the current process will wait to input a value from the process named P, and $Q!out$ represents a desire to output a value out to the process named Q; the sending process is willing to wait till Q is ready to input the value.

Example 5.2 *Assume there is a printer shared by two processes P_1 and P_2. Both processes and the printer are modeled as CSP processes, which together form a "closed" system.*

$$
\begin{array}{llllllllllll}
[& P_1 & :: & \textbf{do} & \neg done_1 & \triangleright & local_{1,1}; PR!e & \textbf{od}; PR!eot; local_{1,2} & & & & \\
\| & P_2 & :: & \textbf{do} & \neg done_2 & \triangleright & local_{2,1}; PR!e; & \textbf{od} PR!eot; local_{2,2} & & & & \\
\| & PR & :: & \textbf{do} & \Box_{i=1}^{2} \textbf{true}; P_i?v & \triangleright & \textbf{do} & v \neq eot & \triangleright & print(v); P_i?v; & \textbf{od} & \textbf{od} \\
] & & & & & & & & & & & &
\end{array}
$$

The printer process PR waits till one of the two processes P_1, P_2, is ready to begin transmission, with the first value. In case both processes want to output to the printer, PR has to make a choice. Having chosen to communicate with one of them, the printer does not serve the other process till the chosen one sends an end-of-transmission (eot) signal. The printer process never terminates since it keeps waiting indefinitely for P_1 or P_2 to communicate with it[11]*. It is possible for one process to monopolize the printer and prevent the other process from ever gaining access.*

[11] This interpretation is at variance with the so-called distributed termination convention that Hoare originally proposed in the language. However we find our interpretation more suitable for server processes. It also illustrates that we are now in an arena where we deal with systems that do not necessarily always terminate. Indeed in concurrent systems, guaranteeing properties such as deadlock-freedom, non-termination and freedom from starvation may be more important.

Each process has its own state and the states of the different processes are disjoint. All changes in state σ_i of a process P_i are due to local assignments or receipt of input from another process. The set of global states defined as

$$State \;=\; \bigotimes_{i=1}^{n} \; State_i$$

is the Cartesian product of the sets of the states of individual processes. where $State_i$ is the set of states of the process P_i. The metavariable $\bar{\sigma}$ denotes the global state and each σ_i stands for the state of process P_i. The labels we use for our LTS consist of the set of possible communications, defined as

$$\begin{array}{lcl} Inputs & = & \{P?v \mid P \text{ is a process name and } v \in \mathcal{V}\} \\ Outputs & = & \{P!v \mid P \text{ is a process name and } v \in \mathcal{V}\} \\ \mathcal{L} & = & Inputs \cup Outputs \cup \{\varepsilon\} \end{array}$$

The label ε signifies local computation that involves no interaction with other processes. λ is a meta-variable that ranges over $\mathcal{L}$.

The semantics of the commands in a process P_i are given in Table 9. We will assume below that $j \neq i$. The *Input* rule says that process P_i attempting to receive a value from process P_j can, on receipt of *any* value v from P_j, bind v to a variable x in its local state. Expression e is evaluated to a value v before the process attempts to send it to P_j, the statement terminating if and when P_j accepts this communication. Assignment is considered an internal action that does not affect other processes, and the transition is labeled with ε. In the rules *Seq* and *Int* we abstract from the internal computations of a process by coalescing local changes of state (labelled with ε) into a single labelled transition. The last rule abstracts from local computations and highlights an interaction, whenever there is one. Observe that the *Int* rules are not syntax-directed.

We now deal with the parallel composition of processes. The transitions of *processes* (as opposed to commands) are also labelled (*e.g.*, $\xrightarrow{\lambda}_p$) and have a subscript p to distinguish them from the transition relation $\longrightarrow$ (used in Table 9) for command transitions.

For readability, we follow the following notational conveniences in Table 10.

- For any global state $\bar{\sigma}$, σ_k will denote the k-th component of the n-tuple $(1 \leq k \leq n)$.
- For each k, $1 \leq k \leq n$, $p_k \equiv P_k :: c_k$ and $p'_k \equiv P_k :: c'_k$.

$$\textit{Input} \quad \frac{}{\langle \sigma_i,\ P_j?x \rangle \xrightarrow{P_j?v} \sigma_i[x \mapsto v]}$$

$$\textit{Output} \quad \frac{\sigma_i \ \vdash \ e \Longrightarrow_e v}{\langle \sigma_i,\ P_j!e \rangle \xrightarrow{P_j!v} \sigma_i}$$

$$\textit{Assign} \quad \frac{\sigma_i \ \vdash \ e \Longrightarrow_e v}{\langle \sigma_i,\ x \ := \ e \rangle \xrightarrow{\varepsilon} \sigma_i[x \mapsto v]}$$

$$\textit{Seq} \quad \frac{\langle \sigma_i,\ c_1 \rangle \xrightarrow{\varepsilon} \sigma_i' \qquad \langle \sigma_i',\ c_2 \rangle \xrightarrow{\varepsilon} \sigma_i''}{\langle \sigma_i,\ c_1; c_2 \rangle \xrightarrow{\varepsilon} \sigma_i'}$$

$$\textit{Int}_1 \quad \frac{\langle \sigma_i,\ c \rangle \ (\xrightarrow{\varepsilon})^* \xrightarrow{\lambda} (\xrightarrow{\varepsilon})^* \ \langle \sigma_i',\ c' \rangle}{\langle \sigma_i,\ c \rangle \xrightarrow{\lambda} \langle \sigma_i',\ c' \rangle} \quad \lambda \neq \varepsilon$$

$$\textit{Int}_2 \quad \frac{\langle \sigma_i,\ c \rangle \ (\xrightarrow{\varepsilon})^* \xrightarrow{\lambda} (\xrightarrow{\varepsilon})^* \ \sigma_i'}{\langle \sigma_i,\ c \rangle \xrightarrow{\lambda} \sigma_i'} \quad \lambda \neq \varepsilon$$

Table 9: Mixed-step semantics for CSP commands

- In rules $Par_{interleave}$ and Par_{sync},

$$S \quad \equiv \quad [\ \|_{k=1}^{n}\ p_k\ ] \quad , \quad S' \quad \equiv \quad [\ \|_{k=1}^{n}\ p_k'\ ]$$

- In rule $Par_{interleave}$,

$$\sigma_k' \quad = \quad \begin{cases} \sigma_i' & \text{if } k = i \\ \sigma_k & \text{otherwise} \end{cases} \quad , \quad c_k' \quad \equiv \quad \begin{cases} c_i' & \text{if } k = i \\ c_k & \text{otherwise} \end{cases}$$

- In rule Par_{sync}

$$\sigma_k' \quad = \quad \begin{cases} \sigma_i' & \text{if } k = i \neq j \\ \sigma_j' & \text{if } k = j \neq i \\ \sigma_k & \text{otherwise} \end{cases} \quad , \quad c_k' \quad \equiv \quad \begin{cases} c_i' & \text{if } k = i \neq j \\ c_j' & \text{if } k = j \neq i \\ c_k & \text{otherwise} \end{cases}$$

$$Process_i \qquad \frac{\langle \sigma_i,\ c_i\rangle \xrightarrow{\lambda} \langle \sigma_i',\ c_i'\rangle}{\langle \sigma_i,\ p_i\rangle \xrightarrow{\lambda}_p \langle \sigma_i',\ p_i'\rangle}$$

$$Par_{interleave} \qquad \frac{\langle \sigma_i,\ p_i\rangle \xrightarrow{\varepsilon}_p \langle \sigma_i',\ p_i'\rangle}{\langle \bar{\sigma},\ S\rangle \xrightarrow{\varepsilon}_p \langle \bar{\sigma}',\ S'\rangle}$$

$$Par_{sync} \qquad \frac{\langle \sigma_i,\ p_i\rangle \xrightarrow{P_j!v}_p \langle \sigma_i',\ p_i'\rangle \qquad \langle \sigma_j,\ p_j\rangle \xrightarrow{P_i?v}_p \langle \sigma_j',\ p_j'\rangle}{\langle \bar{\sigma},\ S\rangle \xrightarrow{\varepsilon}_p \langle \bar{\sigma}',\ S'\rangle}$$

Table 10: Big-step semantics for CSP commands

In Table 10:

- The rule Par_{sync} treats a "closed" system of processes. Hence all communications between components of the system are internal to the system.
- The system of processes terminates only if every process in the system terminates. In other words, configurations of the form $\langle \bar{\sigma},\ [\ \|_{k=1}^{n}\ P_k :: \circ\ ]\rangle$ (where "$\circ$" denotes an empty continuation) are the only terminal configurations.
- If the system reaches a stuck configuration, then it is said to be *deadlocked*. In other words, a configuration $\langle \sigma,\ S\rangle$, which is not terminal and such that $\langle \bar{\sigma},\ S\rangle \not\xrightarrow{\varepsilon}_p$ is deadlocked.

Table 11 contains the rules for guards using yet another labelled transition system, which is then used in giving the semantics of the conditional and iterations constructs. (Table 12).

The following example illustrates some of the distinctions that can arise due to non-determinism.

Example 5.3 *Compare the process PR in Example 5.2 with the following alternative version.*

$$PR' \ ::\ \textbf{do}\ \Box_{i=1}^{2}\ \textbf{true} \ \triangleright\ P_i?v; \textbf{do}\ v \neq eot \ \triangleright\ print(v); P_i?v;\ \textbf{od}\ \textbf{od}$$

The major difference between PR and PR' is in their deadlock behavior. Whereas PR may wait till one of the processes is ready to communicate with

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true}}{\langle \sigma,\ e_j \triangleright c_j \rangle \xrightarrow{\varepsilon}_g \sigma}$$

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true} \quad \langle \sigma,\ io_j \rangle \xrightarrow{\lambda} \sigma'}{\langle \sigma,\ e_j; io_j \rangle \xrightarrow{\lambda}_g \sigma'}$$

Table 11: Mixed step semantics for guards

it, PR' is forced to make a commitment to wait on one of the two processes say P_1, regardless of whether P_1 wants to communicate with it. PR' clearly exacerbates the possibilities of deadlock in the system. Therefore, PR and PR' cannot be considered equivalent as processes.

5.2 Extensions

We conclude this discussion with some language features that can easily be modeled in the framework of LTSs.

Input and output. Commands are extended with input and output primitives:

$$c \ ::= \ \ldots \ \mid \ \mathbf{read}(x) \ \mid \ \mathbf{write}(e)$$

Input and output are really special cases of communication, but instead of interacting with a named process, values are taken from and added to stream data structures. The command level rules are (following the convention mentioned above):

$$\textit{Read} \quad \frac{}{\langle \sigma_i,\ \mathbf{read}(x) \rangle \xrightarrow{?v} \sigma_i[x \mapsto v]}$$

$$\textit{Write} \quad \frac{\sigma_i \vdash e \Longrightarrow_e v}{\langle \sigma_i,\ \mathbf{write}(e) \rangle \xrightarrow{!v} \sigma_i}$$

Two new kinds of labels are added, for reading and writing:

$$l \in \mathcal{L} \ ::= \ \ldots \mid !v \mid ?v$$

$$\frac{\langle \sigma,\ g_j \rangle \ \xrightarrow{\lambda}_g \ \sigma'}{\langle \sigma,\ \mathbf{IF} \rangle \ \xrightarrow{\lambda} \ \langle \sigma',\ c_j \rangle} \qquad (j \in \{1, \ldots, n\})$$

$$\frac{\langle \sigma,\ g_j \rangle \ \xrightarrow{\lambda}_g \ \sigma'}{\langle \sigma,\ \mathbf{DO} \rangle \ \xrightarrow{\lambda} \ \langle \sigma',\ c_j;\ \mathbf{DO} \rangle} \quad (j \in \{1, \ldots, n\})$$

$$\frac{\bigwedge_{i=1}^{n} \sigma \ \vdash \ e_i \ \Longrightarrow_e \ \mathbf{false}}{\langle \sigma,\ \mathbf{DO} \rangle \ \xrightarrow{\varepsilon} \ \sigma}$$

$$Let\ \mathbf{IF} \equiv \mathbf{if} \ \square_{i=1}^{n} g_i \ \triangleright \ c_i \ \mathbf{fi}$$

$$and\ \mathbf{DO} \equiv \mathbf{do} \ \square_{i=1}^{n} g_i \ \triangleright \ c_i \ \mathbf{od}$$

Table 12: Semantics of **if** $-$ **fi** and **do** $-$ **od**

At the global configuration level, (global) input and output streams are added. The labels generated at the command level are "discharged" at the top level, with the corresponding manipulations of the I/O streams ς_i, ς_o

$$Rd \quad \frac{\langle \sigma_i,\ c_i \rangle \xrightarrow{?v} \langle \sigma_i',\ c_i' \rangle}{\langle \sigma_i, p_i, v\varsigma_i, \varsigma_o \rangle \xrightarrow{\varepsilon}_p \langle \sigma_i', p_i', \varsigma_i, \varsigma_o \rangle}$$

$$Wrt \quad \frac{\langle \sigma_i,\ c_i \rangle \xrightarrow{!v} \langle \sigma_i',\ c_i' \rangle}{\langle \sigma_i, p_i, \varsigma_i, \varsigma_o \rangle \xrightarrow{\varepsilon}_p \langle \sigma_i', p_i', \varsigma_i, \varsigma_o v \rangle}$$

Dynamic Process Creation. Consider a command $\mathbf{fork}(P, c)$, which dynamically creates a new process named P executing the command c. At the command level, the effect of this command returns the state unchanged, but generates a new kind of label $\Phi(\langle \sigma_i,\ P :: c \rangle)$. The state σ_i is cloned and packaged into the label.

$$\frac{}{\langle \sigma_i,\ \mathbf{fork}(P, c) \rangle \ \xrightarrow{\Phi(\langle \sigma_i,\ P :: c \rangle)} \ \sigma_i} \quad \text{where } P \text{ is a new process name}$$

At the global configuration level, the label $\Phi(\langle \sigma_i,\ P :: c \rangle)$ is "discharged", by creating a new process with its own local state.

$$\frac{\langle \sigma_i,\ p_i \rangle \xrightarrow{\Phi(P::c)}_p \langle \sigma_i',\ p_i' \rangle}{\langle \bar{\sigma},\ S \rangle \longrightarrow_p \langle \bar{\sigma''},\ S'' \rangle}$$

$S'' = [\ (\|_{k=1}^{n}\ p_k')\ |\ P :: c\ ]$ and $\bar{\sigma''} = \bar{\sigma'} \otimes \sigma_i$, where we continue with the notational convention mentioned above. That is, the vector of process code has a $n+1^{th}$ component $P :: c$ the local state of which has a fresh copy of σ_i as its initial local state. The rule applies only under the assumption that P is a globally fresh process name.

6 Conclusion

We have seen the use of structural operational semantics both as a concise formalism and as a method of precisely defining the dynamic semantics of programming language constructs. The conciseness of the formalism makes it far easier to study and comprehend the potential bottlenecks that an implementor is likely to face. Since the semantics is syntax-driven and the rules are essentially syntactic, it is also possible in many cases, to generate prototypical implementations of new and so far untried constructs quickly with the help of scanning and parsing tools. One such tool for concurrent systems is the Process Algebra compiler of North Carolina [CMS95].

In the case of both imperative and functional languages, we have chosen the semantics of a small core and built up new constructs and features and given them meaning. However, in general, an existing programming language cannot be extended by adding new features to it, without first considering how the existing features of the language interact with the new ones.

In many cases, the implementation strategies become clearer through such a rule-based exposition of the semantics. In certain cases, of course, we have chosen to define rules that are consistent with and model current implementation strategies.

We have not treated the semantics of structured data in general. We have also not treated the semantics of types or static semantic analysis. While this is a major omission and is important for compiling, it would have taken us too far afield. Another significant omission is the semantics of modules, classes and objects much of which is still an area of active research. The bibliography

contains several references which the reader may consult to learn more about the work in the area.

References

[AC98] R. M. Amadio and P.-L. Curien. *Domains and lambda-calculi.* Cambridge University Press, 1998.

[AFV00] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra.* Elsevier, Amsterdam, 2000.

[ANB+86] E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, G. Reggio, and E. Zucca. *The Trial Definition of Ada, Deliverable 7 of the CEC MAP project: The Draft Formal Definition of ANSI/MIL-STD 1815 Ada.* CEC MAP, 1986.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Ast91] E. Astesiano. *Inductive and Operational Semantics*, pages 53–134. Formal Description of Programming Concepts. Springer-Verlag, 1991.

[Bar84] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics.* North Holland, Amsterdam, 1984.

[BC84] G. Berry and L. Cosserat. The **Esterel** synchronous programming language and its mathematical semantics. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer-Verlag, 1984.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BH87] R. Burstall and F. Honsell. A natural deduction treatment of operational semantics. In *Proceedings of FST and TCS 8, Foundations of Software Technology and Theoretical Computer Science, Pune India*, volume 287 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Bor00] Richard Bornat. Proving pointer programs in Hoare Logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[BS90] Egon Borger and Peter H. Schmitt. A formal operational semantics for languages of type prolog III. In *CSL*, pages 67–79, 1990.

[Car84] L. Cardelli. Compiling a functional language. In *Proceedings of 1984 Symposium on LISP and Functional Programmin*, pages 208–217, 1984.

[Car86a] L. Cardelli. Amber. In G. Cousineau, P-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Language*, volume 242 of *LNCS*. Springer, 1986.

[Car86b] L. Cardelli. The amber machine. In G. Cousineau, P-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, volume 242 of *LNCS*. Springer, 1986.

[CCM85] G. Cousineau, P. Curien, and M. Mauny. The Categorical Abstract Machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64, Berlin, 1985. Springer-Verlag.

[CIO00] Cristiano Calcagno, Samin Ishtiaq, and Peter W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In Maurizio Gabbrielli and Frank Pfenning, editors, *Proc. 2nd International Conference on Principles and Practice of Declarative Programming, Montreal, Canada*. ACM, 2000.

[CKRW99] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In *Formal Syntax and Semantics of Java*, pages 157–200, 1999.

[CMS95] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In E. Brinksma, R. Cleaveland, K.G. Larsen, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 153–173. Springer Verlag, 1995.

[Cur91] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, 1991.

[dS92] Fabio Q. B. da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics.* PhD thesis, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, Edinburgh, EH9 3JZ, Scotland, September 1992. Available as LFCS Report Series ECS-LFCS-92-241 or CST-95-92.

[FFKD87] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[Gon88] G. Gonthier. *Sémantiques et Modèles d'Exécution des Langages Réactifs Synchrone; Application à* **Esterel**. Thèse d'informatique, Université d'Orsay, 1988.

[Gri90] Timothy G. Griffin. The formulae-as-types notion of control. In *Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan 1990*, pages 47–57. ACM Press, New York, 1990.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[Gur93] Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.

[Han91] John Hannan. Making abstract machines less abstract. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, volume 523, pages 618–635. Springer-Verlag, Berlin, Heidelberg, and New York, 1991.

[Han94] J. Hannan. Operational semantics-directed compilers and machine architectures. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247, 1994.

[Hen80] P. Henderson. *Functional Programming: Application and Implementation*. Prentice Hall International, 1980.

[Hen88] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, Massachusetts, 1988.

[HJP92] Seif Haridi, Sverker Janson, and Catuscia Palamidessi. Structural operational semantics of AKL. *Future Generation Computer Systems*, 1992.

[HL74] C.A.R. Hoare and P.E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, Englewood Cliffs, 1985.

[HP92] J. Hannan and F. Pfenning. Compiler verification in lf. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE, 1992.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, London, 1987.

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of STACS'87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[KS88] Richard Kennaway and Ronan Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.

[Lan64] P.J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6(5):308–320, 1964.

[Lan65a] P. J. Landin. An abstract machine for designers of computing languages. In *Proc. IFIP Congress*, pages 438–439, 1965.

[Lan65b] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.

[Lau68] L.P. Lauer. Formal definition of Algol 60. Technical Report TR.25.088, IBM Lab. Vienna, 1968.

[Lei01] J. J. Leifer. *Operational Congruences for Reactive Systems*. PhD thesis, University of Cambridge Computer Laboratory, 2001.

[McC63] J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing 1962*, pages 21–28, 1963.

[MH90] D. Miller and J. Hannan. From operational semantics to abstract machines: Preliminary results. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. ACM, 1990.

[Mic94] Marino Miculan. The expressive power of structural operational semantics with explicit assumptions. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 263–290. Springer-Verlag LNCS 806, 1994.

[Mil73] R. Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium 1973,* Bristol, UK, pages 158–173. North-Holland, 1973.

[Mil76] R. Milner. Program semantics and mechanized proof. In K. R. Apt and J. W. de Bakker, editors, *Foundations of Computer Science II*, pages 3–44. Mathematical Centre, Amsterdam, 1976.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[Mor82] J. Morris. A general axiom of assignment and linked data structure. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–41. ??, 1982.

[Mor88] James Morris. *Algebraic operational semantics for Modula 2*. PhD thesis, University of Michigan, 1988.

[Mos92] P. D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[MS96] D. Le Metayer and D. Schmidt. Structural operational semantics as a basis for static program analysis. *ACM Computing Surveys*, 28:340–343, 1996.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Ong99] C.-H. L. Ong. Correspondence between Operational and Denotational Semantics: The Full Abstraction problem for PCF. In S. Abramsky, editor, *Handbook of Theoretical Computer Science*, volume 3. Oxford University Press, 1999.

[Pal92] Jens Palsberg. A provably correct compiler generator. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582, pages 418–434. Springer-Verlag, New York, NY, 1992.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *5th GI Conference,* Karlsruhe, Germany, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[PL/86] PL/I Definition Group. Formal definition of PL/I version 1. Report TR25.071, American Nat. Standards Institute, 1986.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[Plo81] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[Plo83] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II, Garmisch-Partenkirchen*, pages 199–225. North-Holland, 1983.

[San97] David Sands. From SOS rules to proof principles: An operational metatheory for functional languages. In *Conference Record 24th ACM Symposium on Principles of Programming Languages*, pages 428–441, Paris, France, 1997.

[Sch86] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[Sew98] P. Sewell. From rewrite rules to bisimulation congruences. In *Proceedings of CONCUR'98*, volume 1466 of *LNCS*, pages 269–284. Springer Verlag, 1998.

[Sto77] J. Stoy. *Denotational Semantics: the Scott-Strachey approach to Programming Language Theory.* MIT press, 1977.

[Ten81] R. D. Tennent. *Principles of Programming Languages.* Prentice-Hall International, 1981.

[Tin01] Simone Tini. An axiomatic semantics for Esterel. *Theoretical Computer Science*, 269, 2001.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9(1):31–49, 1979.

[War83] D. H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, California, 1983.

[Wat90] D.A. Watt. *Programming Concepts and Paradigms*. Prentice Hall, 1990.

[WBB92] S. Weber, B. Bloom, and G. Brown. Compiling Joy to silicon: A verified silicon compilation scheme. In *Proceedings of the Advanced Research in VLSI and VLSI and Parallel Systems Conference*, Providence, RI, 1992.

[WF94] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An introduction*. Foundations of Computing Science. MIT Press, 1993.

[WO92] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 151–160, New York, 1992.