Principled and Pragmatic Specification of Programming Languages

Adrian Johnstone[0000−0002−9446−9701] and Elizabeth Scott[0000−0001−5907−8513]

Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK
{a.johnstone,e.scott}@rhul.ac.uk

Abstract. Programmers from the imperative tradition often have little experience of using inductive definitions and inference, and that may explain why executable SOS specifications have not become a standard feature of mainstream language development toolkits. We wish to 'demystify' SOS for such programmers, allowing precise and principled specifications to be given for even small industrial DSL's. eSOS (elided Structural Operational Semantics) is a compact tool for specifying executable formal semantics. It is designed to be a translation target for enriched SOS specification languages. The simplicity of eSOS and its reference Java implementation allow programmers to follow the details of an execution trace, and to step through rules using a conventional debugging framework, allowing them to understand and use SOS-based specifications to construct usable language interpreters.

Keywords: Structural Operational Semantics, Domain Specific Language specification, operationalising formal specifications

1 Introduction

Formal specification of programming language semantics is still seen by most software engineers as an esoteric and opaque approach to language implementation. In this paper we describe our approach to 'de-mystifying' formal semantics by embedding a simple model of SOS interpretation into a final year course on the engineering of Domain Specific Languages. We use the formal specification of semantics as a concise and precise specification from which an interpreter may be automatically generated rather than emphasising verification or proving properties of programs. We limit ourselves to sequential languages and as a result direct interpretation of the rules can yield processors which are fast enough for many applications.

A key part of the approach is to show how to write the SOS rule interpreter itself in a few lines of a procedural programming language. This allows practitioner programmers who may have little or no mathematical training to understand operationally how the specification is executed, and to think of formal specification as 'just another kind of programming'. We have found that the approach is successful even with 'maths-averse' students.

Is this sort of programmer-driven approach necessary or even desirable? Well, it is nearly 40 years since Plotkin introduced Structural Operational Semantics (SOS) in a series of lectures at Aarhus University [6]. In that time the computer science theory community has generated myriad related papers, and SOS is now firmly established as part of the basic toolkit for any programming language researcher. Perhaps surprisingly, the practitioner community has in large measure eschewed formal semantics, including SOS. This is in marked contrast to, say, syntax definition using BNF, formal parsing algorithms and even attribute grammars which are widely used (at least in the somewhat informal way that Bison and some other parsing toolkits provide). This ought not to be the case: the core ideas of SOS are certainly no harder to grasp than the notions of grammars and derivation trees.

We would consider that SOS had entered the mainstream if some of the following were true: textbooks on languages and compilers had a chapter on using SOS to write precise descriptions of some or all of a language; widely used language implementation toolkits included a SOS specification capability and an associated interpreter; programming language standards used at least a little SOS to clarify details; and online forums featured discussion of the pragmatics of applying SOS. In fact none of these are true – for instance on the Stack Overflow forums there are a few questions about SOS and its place in the spectrum of formal semantics techniques, but almost none of the pragmatic 'how do I do X in Y' questions that characterise the forums for widely deployed software tools.

Why might this be? It could be that SOS specifications quickly become too large to be useful, but current informal programming language standards documents are hardly noted for brevity. We suspect that the problem simply arises from the usual cultural gap in our discipline: that in practice the entry price for understanding declarative specifications comprising inductive definitions of relations via inference rules is too high for many working procedural programmers.

Our hypothesis, then, is that if we could reduce SOS interpretation to a simple procedural operation over the rules, programmers would embrace the brevity and clarity of the approach. The problem seems to be that the core idea of inference and the heavy use of mathematical notation in typical SOS textbooks are offputting to 'normal' programmers: perhaps ironically, they simply don't understand the meaning of the semantic formalism.

Several ambitious projects aim to deliver the benefits of formal semantics specification in a programmer-friendly manner; including the well known and now-venerable ASF+SDF system [10] (and its successors including RascalMPL [4] and the Spoofax [3] language workbench), the K system [7] and tools such as OTT [9]. The PlanCompS project [1] provides a unifying approach by abstracting away from formal semantics frameworks, building specifications from small *fundamental constructs*.

Our eSOS system, described in this paper, certainly does not compete with these rich systems, but rather attempts to leverage procedural programmers' existing knowledge to give them a way into formal semantics. The core tool comes as two small Java packages, one containing a value system which pro-

vides straightforward runtime type checking, and the other containing a parser for eSOS specifications, classes implementing an abstract syntax for SOS, and an SOS interpreter for sequential programs. Though conceived as a back end for richer SOS notations, we have found that it is a comfortable notation for neophyte users who can think of it as just another form of programming, and can answer questions such as 'yes, but what does that really *do*' by looking at the source code and exercising SOS specifications using the debugger from their preferred development environment to walk through traces.

2 The Course

We run a third year course entitled *Software Language Engineering (SLE)*. This was originally conceived as a pragmatic counterpart to our existing compiler theory course which presents topics in parsing and code generation and optimisation. The SLE course is intended to equip students with the engineering skills needed to design and deliver a fully working interpreter or compiler for a small language, with no emphasis on optimisation: the primary goal is correctness of the language processor, not high performance. The focus is on Domain Specific Languages, with motivating examples which include 3D modelling languages for graphics and 3D printing; music specification languages which connect to the Java MIDI synthesizer; and image processing languages. As part of the course, students develop their own DSL's, usually in one of these domains.

Over time, the two courses have developed independently, and now not all of the SLE students take the compiler theory course. As a result, we work *ab initio* and make no assumptions about prior knowledge of compilers.

The students are rarely mathematically confident. They will have taken typical first year courses on discrete maths with some exposure to logic and the use of inference rules, but they will not have previously applied that knowledge beyond very small pencil-and-paper exercises.

The course is taught over ten weeks (plus one week of revision and consolidation) each of which has two one-hour lectures and a two-hour lab session. There are seven programmed labs; the remaining sessions are used for tutorial support whilst the students develop their own language projects.

The first week is critical. The goal is to de-mystify formal systems by presenting rule based 'symbol-pushing' games. We use Conway's Game of Life as an example. Our students are familiar with this formal system because the first large program that they write in year one is a graphical version of Life.

We then need to help students become comfortable with a reduction model of program execution in which the program is progresively rewritten (with side-effects recorded in semantic *entities* such as the store and the environment). Most students have a von Neumann mind set in which a static program is traversed under the control of the program counter: we tell them that we need to 'get rid of the program counter' before we can use our chosen formal specification method.

We introduce the idea of establishing a program's meaning by repeatedly rewriting it, rather than by an execution walkthrough, using a version of Eu-

clid's Greatest Common Divisor (GCD) algorithm written in a simple procedural language with implicit declarations:

```
a := 15; b := 9;
while a != b
  if a > b
    a := a - b;
  else
    b := b - a;
gcd := a;
```

The program leaves its result in variable `gcd`; with `a` and `b` initialised to 15 and 9 we expect that after execution `gcd` would contain 3.

We also give the program in an internal (abstract) syntax form, as a term built from prefix functions. The internal abstract syntax is not formally defined at this stage: we simply use a form that is sufficiently close to the concrete program that students can accept it as being equivalent.

```
seq(seq(seq(
  assign(a, 15), assign(b, 9)),
  while(ne(deref(a), deref(b)),
    if(gt(deref(a), deref(b)),
      assign(a, sub(deref(a), deref(b))),
      assign(b, sub(deref(b), deref(a))))),
  assign(gcd, deref(a)))
```

We then compare the behaviour of the concrete program (as observed via a walkthrough in the Eclipse debugger) with the behaviour of the internal abstract syntax term under term rewriting. This is a purely illustrative exercise, but nevertheless sufficient to informally show that term rewriting can mimic the execution of the program as conventionally understood.

In the main body of the course, students learn four key techniques: eSOS interpretation; parsing; attribute grammar evaluation; and (limited) term rewriting. In each case, the technique is presented as a formal system, but with an accompanying procedural model rendered in Java code. Often the procedural model presented in lectures is not fully general but is sufficient to provide an intellectual model that allows them to use more powerful versions of the same idea in the labs as a black box, without being burdened by their internal complexity.

Parsing forms a good example of this style of *learn-by-doing, and take-the-rest-on-trust*. For project work, the production parsing technology that we use is the GLL generalised parser [8] but a detailed description of that method would require too much classroom time. Instead, students learn how to hand-write in Java simple singleton-backtracking recursive descent parsers. We then look at grammatical constructs for which which that approach fails: we believe that things which are broken can be more educational than things that seem to magically work. The students go on to use a GLL parser which behaves to some extent like a backtracking parser but overcomes these problems in a way they don't need to know the details of. We extend the parsers to support attributes and attribute equations, and use the resulting tool to implement a grammar for

BNF itself with attribute equations which generate. By the end of this two week segment the students have developed a bootstrapped parser generator which can reproduce itself. They understand parsing, meta-description and generation of programs from specifications. Students know that the parsing technology they have explored is weak, but understand that the principles scale up directly to more general parsing approaches.

3 How we Teach SOS

We teach SOS using eSOS, a variant we have developed to be accessible to mainstream students with a basic procedural programming background. In the later sections we shall describe the eSOS interpreter that allows students to experiment by executing their specifications. However, we don't just want to talk about teaching, we want to illustrate our classroom style. The SLE course has an accompanying textbook which is being developed as we gain experience and feedback from the course. In this section we provide a precis of the lecture material which introduces eSOS to illustrate the way in which we strip the subtleties down to a basic minimum. The rest of Section 3 is written as though for students. This allows the reader who is not a SOS practitioner to pick up the notions and terminology they need for the subsequent sections. Experts will find nothing surprising and can skip to the description of the eSOS interpreter in Section 4.

At hardware level, computer programs exist as essentially static patterns of instructions, traversed under the control of a *program counter* which forms a pointer into the program. It can be difficult to directly prove properties of programs in this model, since the evolution of the computation is a property of the trace of the program counter. An initial step in formalising programming language semantics is often to move to a 'reduction' model, in which the program is a dynamic object that may be rewritten during execution. Most (though not all) execution steps reduce the size of the program term. For pure functional programming languages, these rewrites capture everything there is to say about the computation, but most languages also allow side-effects such as store updates and appends to output lists.

Here is the four step reduction of a program term which computes $10 - 2 - 4$ and 'outputs' the result by appending it to an initially empty list.

```
<output(sub(sub(10, 2),4)), []>
<output(sub(8,4)), []>
<output(4), []>
<, [4]>
```

At each point, some part of the program term called the *reducible expression* or redex has been identified, a simple computation performed and then the term rewritten: in the first step the redex `sub(10, 2)` has been rewritten to `8`.

A configuration ⟨ program term, output list ⟩ thus captures everything about the state of the computation at some point: the list captures side-effects of previous computations and the program term contains what is left to be computed. The output list is an example of a *semantic entity*: depending on the style of language we are specifying, configurations may have several entities in addition to the program term.

In eSOS we have five kinds of entity: (i) read-only lists and (ii) write-only lists model input and output; (iii) maps whose bindings may be changed which model read/write memory (usually called *stores*); (iv) maps whose bindings may not changed which model symbol tables (or *environments*) and (v) singleton sets which are used for describing signals and exceptions. We refer to a ⟨program term, entity list⟩ pair as a *configuration*. If the entity list is empty, we may omit it.

Execution of programs, then, is modeled by stepping from configuration to configuration. The components of a configuration vary according to the language being specified. In the subtraction example we have a program term and a write-only list. Our abstract internal form of the GCD program above does not have input and output, so all we need is a program term and a store, denoted as $\langle \theta, \sigma \rangle$.

We can view program execution as a sequence of configuration transitions: configuration X transitions to Y if there is a program whose transition sequence has Y appearing as a successor to X. The set of all transitions describes everything that could possibly be executed: in a deep sense it *is* the semantics of the language of those programs.

An SOS specification is merely a device for specifying a (usually infinite) set of transitions using a finite recipe of *inference rules*. For languages with configurations $\langle \theta, \sigma \rangle$, each rule has the form

$$\frac{C_1 \quad C_2 \quad \ldots \quad C_k}{\langle \theta, \sigma \rangle \rightarrow \langle \theta', \sigma' \rangle}$$

The single transition below the line is the *conclusion*. The C_i are the *conditions*: there may be zero or more of them. Conditions can themselves be transitions, or may be functions. The latter are referred to as *side-conditions*.

One might read an inference rule in this style as:

> if you have a configuration $\langle \theta, \sigma \rangle$,
> and C_1 succeeds and C_2 succeeds and ... and C_k succeeds
> then we can transition to configuration $\langle \theta', \sigma' \rangle$

One uses this kind of rule by checking that the current configuration matches the left hand side of the conclusion, then checking the conditions (in any order) and then, if everything succeeds, rewriting the current configuration into the right hand side of the conclusion. Where a condition is itself a transition we must recursively apply our checking process to transitions in the conditions. The subchecking can only terminate when we encounter a rule with no transitions in its conditions.

In practice, to produce a finite specification, SOS rules are written as *rule schemas* in which variables are used as placeholders for subterms. For example,

$$< \mathsf{seq}(\mathit{done}, C) > \rightarrow < C >$$

is a rule schema with variable C, and a rule is obtained by replacing C with a program term

$$< \mathsf{seq}(\mathit{done}, \mathsf{output}(6)) > \rightarrow < \mathsf{output}(6) > .$$

When interpreting these rule schemas, we use the operations of *pattern matching* and *substitution* to dissect and reconstruct terms. We call a term which contains variables an *open term* or *pattern*. A term with no variables is *closed*.

We shall write $\theta \triangleright \pi$ for the operation of matching closed term θ against pattern π. The result of such a pattern match is either *failure* represented by $\perp$, or a set of bindings. So, in these expressions where X is a variable

$$\mathsf{seq}(\mathsf{done},\ \mathsf{output}(6)) \triangleright \mathsf{seq}(\mathsf{done},\ X)$$

returns $\{X \mapsto \mathsf{output}(6)\}$ whereas

$$\mathsf{seq}(\mathsf{done},\ \mathsf{output}(6)) \triangleright \mathsf{seq}(X,\ \mathsf{done})$$

returns $\perp$ because $\mathsf{output}(6)$ does not match done.

Pattern substitution is the process of substituting bound subterms for the variables in the pattern. We shall write $\pi \triangleleft \rho$ for the operation of replacing variables in pattern π with their bound terms from ρ. So

$$\mathsf{plus}(X,\ 10) \triangleleft \{X \mapsto 6\} \quad \text{returns } \mathsf{plus}(6,\ 10).$$

The following SOS rule (schema) handles the subtraction of two integers. It has three side conditions which use pre-specified functions `isInt` and `subOp`. The construct `sub` belongs to the abstract syntax of the language whose semantics are being specified.

$$\frac{\mathtt{isInt}(n_1) \triangleright \mathit{true}\ \ \mathtt{isInt}(n_2) \triangleright \mathit{true}\ \ \mathtt{subOp}(n_1, n_2) \triangleright V}{\langle \mathsf{sub}(n_1, n_2) \rangle \rightarrow \langle V \rangle} \qquad [\text{sub}]$$

The conclusion tells us that this rule will rewrite expressions of the form `sub(n1, n2)` to some value, if the conditions (which are all side-conditions) are met.

How should we use such rules to implement interpreters? Let us assume that the current program term is θ, then one way to compute whether the transition may be made is:

if $\rho_1 = (\theta \triangleright \mathsf{sub}(n_1, n_2))$ **then**
  if $(\mathtt{isInt}(n_1) \triangleleft \rho_1) \triangleright \mathit{true}$
    and $(\mathtt{isInt}(n_2) \triangleleft \rho_1) \triangleright \mathit{true}$
    and $\rho_2 = ((\mathtt{subOp}(n_1, n_2) \triangleleft \rho_1) \triangleright V)$
      then $\theta' = V \triangleleft \rho_2$

Informally, we try to match the current program term against the left-hand side of the conclusion and store any variable bindings in the map ρ_1. We then work through the conditions substituting for variables on their left hand sides and perhaps creating new environments for pattern matches. If all of the conditions succeed then we make a new program term θ' by substituting the most recent environment. If we can guarantee that each variable appears only once as an argument to a pattern match operator $\triangleright$, then we can use a single environment which is extended as we work through the conditions.

Using the two rules below, we specify program terms which are nested subtractions.

$$\frac{\langle E_1, \alpha\rangle \rightarrow \langle I_1, \alpha\rangle}{\langle \mathsf{sub}(E_1, E_2), \alpha\rangle \rightarrow \langle \mathsf{sub}(I_1, E_2), \alpha\rangle} \qquad \text{[subLeft]}$$

$$\frac{\langle E_2, \alpha\rangle \rightarrow \langle I_2, \alpha\rangle \quad \mathtt{isInt}(n) \triangleright \mathit{true}}{\langle \mathsf{sub}(n, E_2), \alpha\rangle \rightarrow \langle \mathsf{sub}(n, I_2), \alpha\rangle} \qquad \text{[subRight]}$$

The rule [subLeft] rewrites the left argument to a simpler expression whilst preserving the second argument. Rule [subRight] will only process terms that had a single integer as the left hand argument, and rewrites the second argument. The original [sub] rule will then perform the subtraction of the integers. Together these three rules comprise a so-called *small-step* SOS for subtraction and act so as to enforce left to right parameter evaluation order.

When running the interpreter on a particular initial term, we can put in checks to ensure that at most one rule is activated at each rewrite step, though of course that will only detect non-determinism that is triggered by that particular term. Static checking of rules can detect some forms of non-determinism.

Example: SOS rules for the GCD internal language

An SOS specification may name more than one set of transitions. The rules we have looked at so far are so-called 'small-step' rules. Big-step rules in which, say, arithmetic operations proceed directly to their result without the fine-grained elaboration of the left and right arguments are also possible, and both types of transition may occur within one set of rules. We illustrate this technique with a complete set of rules for our GCD abstact internal language in which the relational and arithmetic operations are specified using a big-step transition $\Rightarrow$ and the commands using a small-step transition $\rightarrow$. It is sometimes helpful to think of small-step rules such as [assignResolve] 'calling' the big step transition to reduce a complex arithmetic expression to a value.

As well as arithmetic and boolean values, this specification uses the special value done (sometimes called `skip` in the literature) which represents the final reduction state of a program term.

$$< \mathsf{seq}(\mathit{done}, C), \sigma > \rightarrow < C, \sigma > \qquad \text{[sequenceDone]}$$

$$\frac{< C_1, \sigma > \rightarrow < C_1', \sigma' >}{< \mathsf{seq}(C_1, C_2), \sigma > \rightarrow < \mathsf{seq}(C_1', C_2), \sigma' >} \qquad \text{[sequence]}$$

$$< \mathsf{if}(\mathit{true}, C_1, C_2), \sigma > \rightarrow < C_1, \sigma > \qquad \text{[ifTrue]}$$

$$< \mathsf{if}(\mathit{false}, C_1, C_2), \sigma > \rightarrow < C_2, \sigma > \qquad \text{[ifFalse]}$$

$$\frac{< E, \sigma > \Rightarrow < E', \sigma' >}{< \mathsf{if}(E, C_1, C_2), \sigma > \rightarrow < \mathsf{if}(E', C_1, C_2), \sigma' >} \qquad \text{[ifResolve]}$$

$$\frac{< \mathsf{if}(E, \mathsf{seq}(C, \mathsf{while}(E, C)), \mathit{done}), \sigma > \rightarrow < C', \sigma' >}{< \mathsf{while}(E, C), \sigma > \rightarrow < C', \sigma' >} \qquad \text{[while]}$$

$$\frac{\mathtt{isInt}(n) \triangleright \mathit{true} \quad \mathtt{updateOp}(\sigma, X, n) \triangleright \sigma_1}{< \mathsf{assign}(X, n), \sigma > \rightarrow < \mathit{done}, \sigma_1 >} \qquad \text{[assign]}$$

$$\frac{< E, \sigma > \Rightarrow < n, \sigma' >}{< \mathsf{assign}(X, E), \sigma > \rightarrow < \mathsf{assign}(X, n), \sigma' >} \qquad \text{[assignResolve]}$$

$$\frac{< E_1, \sigma > \Rightarrow < n_1, \sigma_1 > < E_2, \sigma_1 > \Rightarrow < n_2, \sigma_2 > \mathtt{gtOp}(n_1, n_2) \triangleright V}{< \mathsf{gt}(E_1, E_2), \sigma > \Rightarrow < V, \sigma_2 >} \qquad \text{[gtBig]}$$

$$\frac{< E_1, \sigma > \Rightarrow < n_1, \sigma_1 > < E_2, \sigma_1 > \Rightarrow < n_2, \sigma_2 > \mathtt{neOp}(n_1, n_2) \triangleright V}{< \mathsf{ne}(E_1, E_2), \sigma > \Rightarrow < V, \sigma_2 >} \qquad \text{[neBig]}$$

$$\frac{< E_1, \sigma > \Rightarrow < n_1, \sigma_1 > < E_2, \sigma_1 > \Rightarrow < n_2, \sigma_2 > \mathtt{subOp}(n_1, n_2) \triangleright V}{< \mathsf{sub}(E_1, E_2), \sigma > \Rightarrow < V, \sigma_2 >} \qquad \text{[subBig]}$$

$$\frac{\mathtt{valueOp}(\sigma, R) \triangleright V}{< \mathsf{deref}(R), \sigma > \Rightarrow < V, \sigma >} \qquad \text{[variable]}$$

The result of running the eSOS interpreter with these rules on the input term above is a 30-step reduction of the initial term to the terminating value done, the last four configuations of which are:

```
< seq(done, assign(gcd, deref(a))), sig = { a->3 b->3 } >
< assign(gcd, deref(a)), sig = { a->3 b->3 } >
< assign(gcd, 3), sig = { a->3 b->3 } >
< done, sig = { a->3 b->3 gcd->3 } >
```

Happily, after the final step the store in the final configuration contains a binding from `gcd` to 3.

One can write specifications that are incomplete, but appear to work. The characteristic symptom is that the behaviour of the interpreter is sensitive to the order of the rules. In fact this specification contains nondeterminsim: rules [assign] and [assignResolve] can both trigger if the redex is an integer.

With the ordering shown here, the interpreter prioritises [assign] over [assignResolve] which has the effect of invoking [assignResolve] on complex expressions until they are reduced to an integer, at which point [assign] performs the assignment. If the order of the rules is reversed, the interpreter will loop forever on [assignResolve].

The cure for this class of problem is to ensure that sufficient side-conditions are added to the rules to ensure that at most one rule at a time can be triggered.

4 The eSOS Interpreter

The origins of the eSOS tool lie in providing efficient interpretation of rules for funcons. The software was developed within the PLanCompS project as a sort-of 'assembly language' for SOS rules. The intention was to reduce SOS rule interpretation to a minimalist core, with richer and more expressive forms of specification languages (such as Mosses' CBS notation) being translated down into eSOS before execution. Once developed, we created experimental lab sessions for the SLE course. These were very successful and led to a reworking of the course which put SOS at its centre.

In the literature a variety of notations are used within SOS specifications. Some are just syntactic sugar: for instance a turnstile symbol $\vdash$ may be used in expressions such as $\rho \vdash \langle \theta, \sigma \rangle \rightarrow \langle \theta', \sigma' \rangle$ as shorthand for $\langle \theta, \rho, \sigma \rangle \rightarrow \langle \theta', \rho, \sigma' \rangle$.

More significantly, most authors use standard mathematical notation where possible, and allow computations and function calls to appear directly within transitions. For instance, a rule for subtraction might be written:

$$\frac{n_1 \in \mathbf{Z} \quad n_2 \in \mathbf{Z}}{\langle \mathsf{sub}(n_1, n_2) \rangle \rightarrow \langle n_1 - n_2 \rangle} \qquad \text{[subConcise]}$$

using standard symbols for set membership and the set of integers. The expression in the right-hand side of the conclusion should be read as the arithmetic result of performing subtraction on the substituted variables n_1 and n_2.

These conventions certainly allow for more succinct expression, but can be a little daunting at first encounter, especially the ellision of side conditions into transitions. We might think of them as 'high level' formats which are convenient for the human reader when exercising small example specifications.

The eSOS format is extremely limited, but no less expressive than these richer forms. We can view it as a low level format in which the operations needed for our style of interpretation are explicit. eSOS allows only the three operations: pattern matching, substitution and evaluation of functions from term(s) to term. In fact the substitution operator is automatically applied to the right hand side of all transitions and side conditions, and so never needs to be written. In addition, configurations must be comprised of terms with no embedded functions.

Functions can only appear on the left hand side of side-conditions. The arguments to, and the return value from, a function, must be terms. This means that terms such as the number 67 or the boolean false are represented as trees containing a single node which is labeled with 67 or `false` accordingly.

New values may be computed and inserted into the result of a transition by matching the result of function to a variable, and then binding that variable in the right hand side of a conclusion, as shown in rule [sub] above.

The current eSOS interpreter works greedily in the sense that the first rule that succeeds will be used, and rules are checked in the order that they are written. Within a rule, conditions are checked in strict left to right order. In principle we could also use more sophisticated interpretation strategies that supported non-determinism so as to model concurrency.

eSOS provides a *value system* which has built in dynamic type checking allowing a designer to test parts of their implementation before they have implemented the static semantics of their type system. The system has a fixed set of operations with suggestive names such as `add`, `union` and so on. The value classes are all extensions of class `Value`, which contains a method for each operation. Within `Value`, the operation methods all throw a Java exception. The idea is that the class for, say, `Integer` extends `Value` and implements its own overriding method for each operation that is meaningful on that type. If an operation is called on an inappropriate value (for which is no operation defined) the top level method in `Value` will issue a run time error.

Function calls in eSOS side conditions are almost all direct calls to the functions in the value library; all the interpreter needs to do is to extract the label from a term (which will be an instance of a value package class) and call the corresponding method. The interpreter contains a case statement which branches on the function name and performs the extract-and-call action. Here is the branch for the `subOp()` function used in our GCD rules:

```
case "subOp":
  functionResult = new ValueTerm(
    leftPayload.sub(children.get(1).getPayload()));
  break;
```

The value system also provides a set of coercion operations which can interconvert values where appropriate.

Most of the value classes are really wrappers on the underlying Java API class. We offer these primitive types: Boolean, Character, Integer32, IntegerArbitrary, Real64, RealArbitrary, Null and Void; and these collection types: Array, String, List, Set, Tuple, Map, Record, MapHierarchy.

The IntegerArbitrary and RealArbitrary classes support arbitrary length values. The MapHierarchy class contains a map and a reference to another MapHierarchy called the parent. If a key lookup is performed on a MapHierarchy, the search proceeds recursively through the base MapHierachy and its parents. This naturally implements nested scoping of key-value bindings. In addition there are Term and TermVariable classes that construct trees whose nodes are labeled with instances of Value types. The Term class includes pattern match and substitute operations. Some of the collection classes also have implementations of match and substitute that generalise over the terms held in the collection.

The implementation of eSOS relies heavily on the classes in the value package; for instance SOS configurations are represented by instances of the Record class and environments by instances of MapHierarchy. Terms are, of course, represented by instances of value class Term and the builtin matching and substitution methods are sensitive to instances of variables represented with the TermVariable class.

With so much of the work being done within the operation methods of the Value library, the main interpreter function may be compactly expressed. The current implementation requires some 30 lines of Java.

5 The eSOS Concrete Syntax

eSOS rules may be constructed directly by programs written in Java and other JVM languages through an Application Programmer Interface (API), but the usual way to create a specification is via a text file containing eSOS concrete rules. The prelude and a concrete form of the first two rules from our GCD specification is shown below. From this, LaTeX source to typeset the equations is automatically generated.

```
relation ->, sig:map, done
relation =>, sig:map
latex sig "\\sigma", -> "\\rightarrow", => "\\Rightarrow"

-sequenceDone
---
seq(done, C) -> C

-sequence
C_1 -> C_1'
---
seq(C_1, C_2) -> seq(C_1', C_2)
```

The `relation` directive declares each transition symbol and zero or more associated syntactic entities. These are typed as one of the five classes of entity mentioned on page 6; in this case entity `sig` is of type `map` and thus can be used to model the store. The configurations of the complete specification is the union of the entities declared in all of the `relation` directives.

The `latex` directive creates a set of mappings which are used to generate LaTeX aliases, enabling us to write `sig` in the source file and have it appear as σ in the typeset output.

The rules themselves are *elided* in that entities which are used in 'standard' ways need not be mentioned. This approach is inspired by Peter Mosses' work on MSOS [5], in which semantic entities are gathered into a record which labels the transition. Mosses provides a category-theoretic classification of propagation rules for entities. In eSOS we use a single propagation rule which we call the 'round the clock' rule, so for instance an unmentioned store entity σ propagates as:

$$\frac{\langle , \sigma_0\rangle \rightarrow \langle , \sigma_1\rangle \ \langle , \sigma_1\rangle \rightarrow \langle , \sigma_2\rangle \ \ldots \langle , \sigma_{k-1}\rangle \rightarrow \langle , \sigma_k\rangle}{\langle , \sigma_0\rangle \rightarrow \langle , \sigma_k\rangle}$$

Apart from reducing the amount of writing required, the main purpose of this elision is to support modularity, allowing fragments of specifications which may use different configurations to be brought together in the manner of MSOS. Our uniform propagation rule has the merit of simplicity but in general will generate more bindings during interpretation than strictly necessary.

Space precludes a detailed example, but the motivation for adopting this capability is to support the Funcon methodology mentioned in Section 1. In

particular, we wish to support the use of signal entities which manage the propagation of exceptions and other forms of unusual control flow. In general, the only constructs needing to access signal entities are those originating or handling the exceptions. We do not want to clutter all of the other rules with references to signals; in eSOS they can be simply elided away in the source, and will then be automatically generated as the rules are expanded for interpretation.

6 Connecting Parsers to eSOS

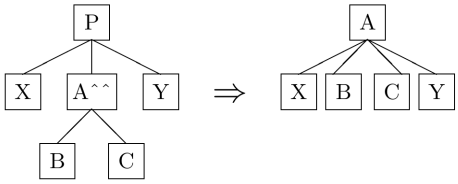
A BNF context free grammar for the GCD language in Section 2 is shown below. Terminals are stropped `'thus'` and we assume the availability of two lexical items `INTEGER` and `ID` which match decimal digit sequences and alpha-numeric identifiers in the conventional way. (Ignore for the moment the `^` annotations.) For compactness, the grammar only provides definitions for the $>$, $\neq$ and subtraction operations though it does encode their relative priorities and associativities. The grammar does not generate empty programs.

```
statement ::= seq^^ | assign^^ | if^^ | while^^
seq ::= statement statement
assign ::= ID ':='^ subExpr ';'^
if ::= 'if'^ relExpr statement 'else'^ statement
while ::= 'while'^ relExpr statement
relExpr ::= subExpr^^ | gt^^ | ne^^
gt ::= relExpr '>'^ subExpr
ne ::= relExpr '!='^ subExpr
subExpr ::= operand^^ | sub^^
sub ::= subExpr '-'^ operand
operand ::= deref^^ | INTEGER^^ | '('^ subExpr^^ ')'^
deref ::= ID
```

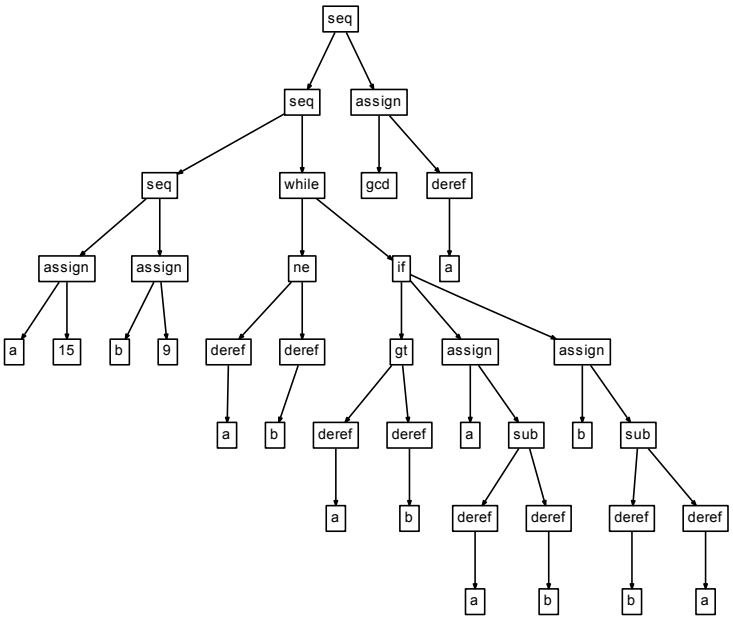
When used to parse the GCD program above, this grammar yields a derivation tree containing 92 nodes. The relatively large structure contains nodes representing, for instance, keywords and punctuation that may be safely discarded without losing the underlying meaning of the program. It is conventional in formal semantics work (and indeed in compiler construction) to generate a more compact intermediate form. For instance, the GNU compilers use the GENERIC libraries to build simplified trees which are translated into three-address code for optimisation, and the metamodelling community typically use Java classes to represent semantic entities which are initialised by concrete parsers.

In formal semantics, connections to concrete parsing are often eschewed in favour of starting with some abstract syntax capturing syntactic-categories such as declarations, commands, expressions and so on. This is reasonable for research, but can be a bar to progress for those wishing to simply execute semantic specifications, whether on paper or via interpreters. For example, how should phrases in the simplified abstract syntax to be constructed from a concrete program source?

An approach we have found useful is to deploy *folds* [2] to convert full derivation trees to simplified abstract syntax trees. There are two fold operations denoted by `^` (fold under) and `^^` (fold over). In both cases, the annotated node A is combined with its parent P and the children of A are 'pulled up' and inserted as children of P, in order, between the siblings of A. When folding-over, the label of P is replaced by the la



When folding under, P retains its original label and thus A disappears: a fold-under applied to a terminal, therefore, has the effect of deleting it from the tree and can be used to remove syntactic clutter such as the `'('` and `')'` terminals in the GCD grammar. Fold-overs can be used to telescope chains of nonterminals: for instance we use it above to overwrite all instances of nonterminal `operand` with `deref`, `subExpr` or an integer literal as appropriate. We have also used carrier nonterminals such as `ge` and `sub` to replace concrete syntax operators such as `>=` with alphanumeric names. The reader may like to check that the annotations above, when applied to the derivation tree for our GCD program yields this abstracted tree, which has 39 nodes.



If we output the labels in a preorder traversal using the usual bracketing convention, we get this text rendition which is in a format suitable for use directly as a program term with the eSOS interpreter.

```
seq(seq(seq(assign(a, 15), assign(b, 9)),
  while(ne(deref(a), deref(b)), if(gt(deref(a), deref(b)),
```

```
    assign(a, sub(deref(a), deref(b))),
    assign(b, sub(deref(b), deref(a)))))),
  assign(gcd, deref(a)))
```

Tree construction with fold operations may be described using an L-attributed grammar and hence folded derivation trees may be produced in a single pass, or even 'on the fly' by recursive descent parsers.

7 Student Response and Conclusions

eSOS is a distillation of the core operating principles of a sequential SOS interpreter, and as such it represents a 'lowest common denominator' of the various enriched notations that one encounters in the research literature. The simple syntax, combined with a compact interpreter written in Java provide a comfortable entry to formal semantics for undergraduate students. Student response has been enthusiastic. The even split between laboratory and lecture room time enabled impressive project work, and in formal questionnaire returns students rated the course as being significantly more intellectually stimulating than the mean scores across all other courses in our school.

References

1. M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. *Trans. Aspect-Oriented Software Development*, 12:132–179, 2015.
2. A. Johnstone and E. Scott. Tear-Insert-Fold grammars. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 6:1–6:8, New York, NY, USA, 2010. ACM.
3. L. C. Kats and E. Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, Oct. 2010.
4. P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 168–177, 2009.
5. P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *Electr. Notes Theor. Comput. Sci.*, 229(4):49–66, 2009.
6. G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
7. G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
8. E. Scott and A. Johnstone. GLL syntax analysers for EBNF grammars. *Science of Computer Programming*, 166:120–145, 11 2018.
9. P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.
10. M. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.