Formal Programming Language Semantics note 9

Extended example: a fragment of Java

In this note we will present a complete static and dynamic semantics for a small fragment of Java — this will show how the techniques of operational semantics scale up to a slightly more ambitious example. It will also serve to illustrate a few points not covered by the languages in earlier notes — in particular:

- The treatment of complex data structures (e.g. a heap of objects pointing at one another) rather than just primitive data values like integers.
- The treatment of methods and method calls (similar ideas can be used to describe functions and procedures in imperative or functional languages).
- The treatment of *recursive* declarations (in this case, recursive methods and recursive classes).

Syntax of COOL. Our language (which we will call **COOL**[1]) will be a small fragment of Java. We claim that *every* program of **COOL** is a legal program of Java, and that for all such programs, the behaviour specified by our semantics agrees with the behaviour specified by the definition of Java. However, we have imposed some quite strong (and silly) restrictions on programs in order to keep our definition reasonably small. E.g.:

- We make do with a single basic type `int`.
- We omit many features that we have seen several times already, like conditionals and `while` loops.
- Each method must take exactly one argument, and must return a value of some type (no `void` return types).
- All field and method access expressions must have an explicit target object, so e.g. one must write `this.foo` rather than just `foo`.
- No field initializers or explicit constructors: we just have the default zero-argument constructor which sets up the fields with their default values.
- All fields are `private`, all methods are `public`.
- No inheritance or overriding.

[1] An acronym for *cut-down object oriented language*. Or maybe a recursive acronym for *cool object oriented language*.

- No overloading of methods.
- No `static` fields or methods.

. . . plus lots of other more trivial restrictions that you may notice. Most of these omitted features could be described easily enough using similar techniques (indeed, you might enjoy working out suitable semantic rules for some of these features).

We do allow *recursive* methods and recursive class definitions in our language, as there is some interest in seeing how these features are described.

As before, we assume we are given a set $\mathbb{I}$ of identifiers, which can be used as names for fields, methods, parameters, or classes. For the purpose of the context-free grammar, we will write $name$ for the lexical category of identifiers. We will sometimes add a subscript (e.g. $name_{field}$) to indicate what role the name is playing.

The syntax of our language is as follows:

$$
\begin{array}{rcl}
\textit{type} & ::= & \texttt{int} \mid name_{class} \\
\textit{expr} & ::= & n \mid \textit{expr} - \textit{expr} \mid \texttt{this} \mid name_{param} \mid \textit{expr}\,.\,name_{field} \mid \\
 & & \textit{expr}\,.\,name_{meth}\,(\textit{expr}) \mid \texttt{new}\ name_{class}\,() \\
\textit{com} & ::= & \textit{expr}\,.\,name_{field} = \textit{expr} \mid \textit{com}\ ;\ \textit{com} \\
\textit{field-decl} & ::= & \texttt{private}\ \textit{type}\ name_{field}\ ; \\
\textit{field-decls} & ::= & \epsilon \mid \textit{field-decls}\ \textit{field-decl} \\
\textit{method-decl} & ::= & \texttt{public}\ \textit{type}\ name_{meth}(\textit{type}\ name_{param})\{\textit{com}\ ;\ \texttt{return}\ \textit{expr}\} \\
\textit{method-decls} & ::= & \epsilon \mid \textit{method-decls}\ \textit{method-decl} \\
\textit{class-decl} & ::= & \texttt{class}\ name_{class}\{\textit{field-decls}\ \textit{method-decls}\} \\
\textit{class-decls} & ::= & \epsilon \mid \textit{class-decls}\ \textit{class-decl} \\
\textit{program} & ::= & \textit{class-decls}\ \textit{expr}
\end{array}
$$

Note that a complete *program* consists of a sequence of class declarations followed by a "top-level" expression to be evaluated (which might be something like a call to a `main` method). Without this top-level expression, there is no way to force any of the code to be executed.

We write Expr and Com for the syntactic categories of expressions and commands respectively.

Static semantics Let's write $\mathbb{U}$ for the set of possible type names. It will also be useful to have a set $\mathbb{U}^+$ containing the additional symbol `none`.

$$\mathbb{U} = \mathbb{I}_{class} \sqcup \{\texttt{int}\}, \qquad \mathbb{U}^+ = \mathbb{U} \sqcup \{\texttt{none}\}$$

For the purpose of the static semantics, a *field environment* will be a finite list of pairs associating types to field names; this is meant to record information about

the fields occurring within a single class. Likewise, a *method environment* will associate parameter and result types to method names; it will also be convenient to record the method body itself (the parameter name, a command, and an expression for the returned value). A *class environment* associates to each class name a corresponding field and method environment.[2] Finally, a *parameter environment* records the name and type of a single method parameter; we also include the dummy parameter environment `none` for use with top-level expressions (where there is no method parameter around).

$$\begin{aligned}
\mathbb{FE} &= (\mathbb{I}_{field} \times \mathbb{U})^* \\
\mathbb{ME} &= \mathbb{I}_{meth} \rightharpoonup (\mathbb{U}_{param} \times \mathbb{U}_{result} \times \mathbb{I}_{param} \times \mathsf{Com} \times \mathsf{Expr}) \\
\mathbb{CE} &= \mathbb{I}_{class} \rightharpoonup \mathbb{FE} \times \mathbb{ME} \\
\mathbb{PE} &= (\mathbb{I}_{param} \times \mathbb{U}) \sqcup \{\texttt{none}\}
\end{aligned}$$

We use FE, ME, CE, PE as variables ranging over $\mathbb{FE}, \mathbb{ME}, \mathbb{CE}, \mathbb{PE}$ respectively. We write CE_F and CE_M for the partial functions $\mathbb{I} \rightharpoonup \mathbb{FE}$, $\mathbb{I} \rightharpoonup \mathbb{ME}$ obtained from $\mathbb{CE}$ using the two projection functions. We will also use f, m, c, p to range over the *names* of fields, methods, classes and parameters respectively. As before we use the notation $FE(\!|f|\!)$ for looking up the information associated with a given name in a given field environment. We will write $\mathrm{dom}\ FE$ for the set of names f such that $FE(\!|\,f\,|\!)$ is defined, and $\mathrm{dom}\ ME$, $\mathrm{dom}\ CE$ for the genuine domains of the partial functions ME, CE.

The "background" against which a particular expression or command should be typechecked will therefore consist of: a class environment CE (giving details of all the classes in scope and their fields and methods); a parameter environment PE specifying the type of the method parameter if any; and an element $\theta \in \mathbb{U}^+$ specifying the current class if any (that is, the type of the current object `this`). For top-level expressions, both PE and θ will be `none`.

We thus expect to have typing assertions of the forms:

$$CE, PE, \theta \vdash \mathit{expr} : u \qquad CE, PE, \theta \vdash \mathit{com} : \texttt{com}$$

The typing rules for expressions are as follows.

$$\frac{}{CE, PE, \theta \vdash n : \texttt{int}}$$

$$\frac{CE, PE, \theta \vdash \mathit{expr}_1 : \texttt{int} \qquad CE, PE, \theta \vdash \mathit{expr}_2 : \texttt{int}}{CE, PE, \theta \vdash \mathit{expr}_1 \texttt{ - } \mathit{expr}_2 : \texttt{int}}$$

[2] We have chosen to take field environments to be lists of pairs, and method and class environments to be genuine partial functions. This reflects the fact that, in Java, the order in which field declarations appear within a class declaration can sometimes make a semantic difference, (think about the effect of adding *field initializers* to **COOL**, for instance), whereas the order of method declarations within a class declaration makes no difference, nor does the order of class declarations within a source file.

$$\frac{}{CE, PE, \theta \vdash \texttt{this} : \theta} \quad \theta \in \mathbb{I}_{class}$$

$$\frac{}{CE, PE, \theta \vdash p : u} \quad PE = (p, u)$$

$$\frac{CE, PE, \theta \vdash \mathit{expr} : u_{target}}{CE, PE, \theta \vdash \mathit{expr}.f : u_{field}} \quad \begin{array}{l} u_{target} = \theta, \\ CE_F(u_{target})(\!|f|\!) = u_{field} \end{array}$$

$$\frac{CE, PE, \theta \vdash \mathit{expr}_1 : u_{target} \qquad CE, PE, \theta \vdash \mathit{expr}_2 : u_{param}}{CE, PE, \theta \vdash \mathit{expr}_1.m\,(\mathit{expr}_2) : u_{result}} \quad \begin{array}{l} CE_M(u_{target})(m) = \\ (u_{param}, u_{result}, -, -, -) \end{array}$$

$$\frac{}{CE, PE, \theta \vdash \texttt{new}\ c\texttt{()} : c} \quad c \in \mathrm{dom}\ CE$$

The typing rules for commands are:

$$\frac{CE, PE, \theta \vdash \mathit{expr}_1 : u_{target} \qquad CE, PE, \theta \vdash \mathit{expr}_2 : u_{field}}{CE, PE, \theta \vdash \mathit{expr}_1\ .\ f\ =\ \mathit{expr}_2\ : \texttt{com}} \quad \begin{array}{l} u_{target} = \theta, \\ CE_F(u_{target})(\!|f|\!) = u_{field} \end{array}$$

$$\frac{CE, PE, \theta \vdash \mathit{com}_1 : \texttt{com} \qquad CE, PE, \theta \vdash \mathit{com}_2 : \texttt{com}}{CE, PE, \theta \vdash \mathit{com}_1\ \texttt{;}\ \mathit{com}_2 : \texttt{com}}$$

We also require rules to say how the various kinds of environments arise. Because of the possibility of recursion, we will first give some rules showing how sequences of declarations give rise to environments under the (seemingly circular) assumption that we already have environments in which to the declarations are to be typechecked. More precisely, we will give rules for assertions of the form

$$CE \vdash \mathit{fds} \Rightarrow FE \qquad CE, \theta \vdash \mathit{mds} \Rightarrow ME \qquad CE \vdash \mathit{cds} \Rightarrow CE'$$

which we may read as saying "relative to CE, the sequence of field declarations *fds* gives rise to the field environment FE", etc. (For typographical convenience we write *fds* in place of *field-decls*, etc.)

The rules for field declarations are fairly trivial:

$$\frac{}{CE \vdash \epsilon \Rightarrow [\,]}$$

$$\frac{CE \vdash \mathit{fds} \Rightarrow FE}{CE \vdash \mathit{fds}\ \ \texttt{private}\ u\ f\ \texttt{;}\ \Rightarrow\ FE; (f, u)} \quad f \notin \mathrm{dom}\ FE$$

The rules for method declarations are more complex:

$$\frac{}{CE, \theta \vdash \epsilon \Rightarrow \emptyset}$$

$$\frac{CE, \theta \vdash \textit{mds} \Rightarrow ME \qquad CE, PE, \theta \vdash \textit{com} : \texttt{com} \qquad CE, PE, \theta \vdash \textit{expr} : u_{result}}{CE, \theta \vdash \textit{mds}\ \texttt{public}\ u_{result}\ m\ (u_{param}\ p)\ \{\textit{com}\ \texttt{;}\ \texttt{return}\ \textit{expr}\} \Rightarrow ME;\ [m \mapsto (u_{param}, u_{result}, p, \textit{com}, \textit{expr})]} \quad \begin{array}{l} PE = (p, u_{param}), \\ m \notin \mathrm{dom}\ ME \end{array}$$

The rules for class declarations are:

$$\frac{}{CE \vdash \epsilon \Rightarrow \emptyset}$$

$$\frac{CE \vdash \textit{cds} \Rightarrow CE' \qquad CE \vdash \textit{fds} \Rightarrow FE \qquad CE, c \vdash \textit{mds} \Rightarrow ME}{CE \vdash \textit{cds}\ \texttt{class}\ c\ \{\textit{fds}\ \textit{mds}\} \Rightarrow CE'[c \mapsto (FE, ME)]}\ c \notin \mathrm{dom}\ CE'$$

Finally, we introduce an assertion form *program* $\rightsquigarrow CE, u$, saying that a program is well-typed and gives rise to the class environment CE, and its top-level expression has type u relative to this class environment. This assertion form is subtly different from those above, in that it does not have the same apparently circular character — no class environment is necessary on the left hand side. The following elegant rule takes care of all forms of recursive declaration in a single swoop; the trick is the double occurrence of CE in the first premise. (It may require a bit of thought to satisfy yourself that this has the desired effect.)

$$\frac{CE \vdash \textit{cds} \Rightarrow CE \qquad CE, \texttt{none}, \texttt{none} \vdash \textit{expr} : u}{\textit{cds}\ \textit{expr} \rightsquigarrow CE, u}$$

It is easy to check that for any *cds* and *expr*, there is at most one pair (CE, u) such that *cds expr* $\rightsquigarrow CE, u$.

Dynamic semantics In order to give a dynamic operational semantics for **COOL**, we need a mathematical model for data in this language. Let $\mathbb{L}$ be some infinite set of abstract *locations*, which we may loosely think of as playing the role of potential memory addresses at which an object may be stored. We let ℓ range over $\mathbb{L}$. We also let $\mathbb{L}^+ = \mathbb{L} \sqcup \{\texttt{none}\}$.

An expression of object type will in general evaluate to a reference to some object (i.e. an element of $\mathbb{L}$), or perhaps the special value `null` (which is not in $\mathbb{L}$ and which plays a different role from `none`). an expression of integer type will of course evaluate to an integer. We therefore define a set $\mathbb{V}$ of potential *values* (ranged over by v):

$$\mathbb{V} = \mathbb{Z} \sqcup \mathbb{L} \sqcup \{\texttt{null}\}$$

The actual contents of an object may be modelled by a *record* associating values to finitely many field names. We write $\mathbb{R}$ for the set of records, and let r range over $\mathbb{R}$. A *heap* may then be modelled as a partial function associating a record to certain locations; this corresponds to giving details of the object stored at a certain address in memory. It is convenient also to include information

giving the class of each object, so that objects "know what their class is" at runtime. We write $\mathbb{H}$ for the set of heaps, and let h range over $\mathbb{H}$.

$$\mathbb{R} \;=\; (\mathbb{I}_{field} \times \mathbb{V})^*, \qquad \mathbb{H} \;=\; \mathbb{L} \rightharpoonup (\mathbb{I}_{class} \times \mathbb{R})$$

For the purpose of recording the values of method parameters at runtime, we also need a set $\mathbb{P}$ of *(dynamic) parameter environments*, ranged over by $\wp$:

$$\mathbb{P} \;=\; (\mathbb{I}_{param} \times \mathbb{V}) \sqcup \{\texttt{none}\}$$

The "background" against which an expression is evaluated or a command executed consists of: a heap $h \in \mathbb{H}$ corresponding to the initial state of memory; a parameter environment $\wp \in \mathbb{P}$ giving the value of the current parameter (if any), and a location $\ell \in \mathbb{L}^+$ giving the location of the current object `this` (if any). (Details of the contents of the current object can then be found by looking up $h(\ell)$.) Both expressions and commands may have an effect on the heap, and the evaluation of expressions will additionally result in a value. We therefore expect our evaluation statements to have the form

$$h, \wp, \ell \vdash \mathit{expr} \Downarrow v, h' \qquad\qquad h, \wp, \ell \vdash \mathit{com} \Downarrow h'$$

We also need some information carried over from the static semantics. Suppose we wish to define the runtime behaviour of a complete program called *program*. We write CE for the class environment arising from the static semantics, i.e. such that $\mathit{program} \rightsquigarrow CE, u$ for some u. Since CE will remain constant throughout the execution of a program, we will not bother to mention it in all our rules, but strictly speaking we should regard the symbol $\vdash$ as an abbreviation for $\vdash_{CE}$.

Finally, some machinery to help us construct records corresponding to newly created objects (rather lazily taking advantage of the fact that constructors in **COOL** can't do anything interesting). To each type $u \in \mathbb{U}$ we may associate a value $\mathrm{default}(u)$ as follows:

$$\mathrm{default}(\texttt{int}) \;=\; 0, \qquad \mathrm{default}(c) \;=\; \texttt{null}$$

To any field environment $FE = [(f_1, u_1), \ldots, (f_k, u_k)]$ we may then associate the default record

$$\mathrm{Default}\ (FE) \;=\; [(f_1, \mathrm{default}(u_1)), \ldots, (f_k, \mathrm{default}(u_k))]$$

The rules for evaluating expressions are now as follows:

$$\frac{}{h, \wp, \ell \vdash n \Downarrow n, h}$$

$$\frac{h_0, \wp, \ell \vdash \mathit{expr}_1 \Downarrow n_1, h_1 \qquad h_1, \wp, \ell \vdash \mathit{expr}_2 \Downarrow n_2, h_2}{h_0, \wp, \ell \vdash \mathit{expr}_1 \ \texttt{-}\ \mathit{expr}_2 \Downarrow n, h_2} \;\; n = n_1 - n_2$$

$$\frac{}{h, \wp, \ell \vdash \texttt{this} \Downarrow \ell, h} \;\; \ell \in \mathbb{L}$$

$$\frac{}{h, \wp, \ell \vdash p \Downarrow v, h} \quad \wp = (p, v)$$

$$\frac{h_0, \wp, \ell \vdash \mathit{expr} \Downarrow \ell', h_1}{h_0, \wp, \ell \vdash \mathit{expr}\,.\,f \Downarrow v, h_1} \quad \begin{array}{l} h_1(\ell') = (-, r), \\ r(f) = v \end{array}$$

$$\frac{\begin{array}{ll} h_0, \wp, \ell \vdash \mathit{expr}_1 \Downarrow \ell', h_1 & h_1, \wp, \ell \vdash \mathit{expr}_2 \Downarrow v, h_2 \\ h_2, \wp', \ell' \vdash \mathit{com} \Downarrow h_3 & h_3, \wp', \ell' \vdash \mathit{expr}_3 \Downarrow v', h_4 \end{array}}{h_0, \wp, \ell \vdash \mathit{expr}_1\,.\,m\,(\mathit{expr}_2) \Downarrow v', h_4} \quad \begin{array}{l} h_1(\ell') = (c, -) \\ CE_M(c)(m) = \\ \quad (-, -, p, \mathit{com}, \mathit{expr}_3) \\ \wp' = (p, v) \end{array}$$

$$\frac{}{h_0, \wp, \ell \vdash \texttt{new}\ c\texttt{()} \Downarrow \ell', h_1} \quad \begin{array}{l} \ell' \notin \mathrm{dom}\ h_0, \quad h_1 = h_0[\ell' \mapsto r] \\ r = \mathrm{Default}\ (CE_F(c)) \end{array}$$

The rules for commands are:

$$\frac{h_0, \wp, \ell \vdash \mathit{expr}_1 \Downarrow \ell', h_1 \qquad h_1, \wp, \ell \vdash \mathit{expr}_2 \Downarrow v, h_2}{h_0, \wp, \ell \vdash \mathit{expr}_1.f = \mathit{expr}_2 \ \Downarrow h_3} \quad \begin{array}{l} h_2(\ell') = (c, r) \\ r' = r[f \mapsto v] \\ h_3 = h_2[\ell' \mapsto (c, r')] \end{array}$$

$$\frac{h_0, \wp, \ell \vdash \mathit{com}_1 \Downarrow h_1 \qquad h_1, \wp, \ell \vdash \mathit{com}_2 \Downarrow h_2}{h_0, \wp, \ell \vdash \mathit{com}_1 \ ; \ \mathit{com}_2 \Downarrow h_2}$$

The final rule allows us to derive evaluation statements $\mathit{program} \Downarrow v$ giving the result of evaluating (the top-level expression of) a complete program. Note that this rule mixes static and dynamic assertions, and we have made explicit the dependency of the dynamic assertions on CE:

$$\frac{\mathit{cds}\ \mathit{expr} \rightsquigarrow (CE, u) \qquad \emptyset, \texttt{none}, \texttt{none} \vdash_{CE} \mathit{expr} \Downarrow v}{\mathit{cds}\ \mathit{expr} \Downarrow v}$$

This completes the formal definition of **COOL**. One can now, for instance, give a precise statement of (a modest version of) *type safety* for this language as follows:

> **Theorem.** If $\mathit{program} \rightsquigarrow (CE, \texttt{int})$ and $\mathit{program} \Downarrow v$, then $v \in \mathbb{Z}$.

Miscellaneous remarks

1. Our language allows expressions that attempt to dereference a null pointer. Our semantics reflects the fact that such expressions do not evaluate successfully by simply failing to generate evaluation statements for them; thus, we have not distinguished between this kind of failure and failure due to non-termination, for instance. It would be a straightforward exercise to incorporate `NullPointerException`s into the language using ideas from Note 5.

2. You will notice that the static semantics of **COOL** is more complicated than the dynamic semantics. This is actually fairly typical for formal descriptions of modern typed languages.

3. Both our static and dynamic semantics are (I think) just about within reach of what could be "animated" using present-day tools. That is, one could provide an animation tool with a more-or-less direct representation of the semantic rules,[3] supply a program to be typechecked or executed, and watch it fly! Indeed, the same tool could typically be used for both static and dynamic semantics, since such tools are usually just engines for general "proof search" that work with a given bunch of rules.

   Although such tools (currently) yield only an inefficient execution of a program, they could still be useful for various purposes, e.g. allowing a language designer to experiment with the effects of various semantic choices; helping to debug a formal semantics for an existing language; or allowing a compiler writer to test the conformance of his compiler to a formal semantics which is regarded as standard.

   Unfortunately, most existing animators of this kind are cumbersome and hard to use. I am still hoping to get round to building a "lightweight" tool that is easy to play with and can cope with languages such as **COOL**.

4. This concludes the operational semantics part of the course. The techniques we have covered are in practice sufficient to allow us to describe almost all features that arise in *sequential* programming languages, and furthermore they are workable for full-scale languages (although some of the rules do sometimes get a bit scary).

   Curiously, one feature which is surprisingly hard to describe using these techniques is unrestricted jumps (i.e. `goto`). This is largely because `goto` doesn't "respect" the syntactic structure of programs but allows you to jump to a random point in the middle of a program phrase. The difficulty in describing this feature formally is of course related to a difficulty in reasoning about it, and one can see this as correlated to the reason why `goto` is frowned upon these days by most sophisticated people. In general, if a language feature is hard to describe formally, this can be seen as some sort of evidence that it is perhaps *inherently* complicated, and so should perhaps be avoided if we want to know that our programs possess good properties.

   The topic of *concurrency* (e.g. threads in Java) is another ball-game altogether, and various process calculi such as CCS (Calculus of Communicating Systems) have been developed to provide mathematical models for it. Another way of modelling concurrency within the framework we have presented is illustrated in Question 1 from the 2004 FPLS exam.[4]

John Longley

[3] A little additional code may be needed to help out with side conditions in some cases.

[4] This turned out to be rather too hard as an exam question, but may provide an entertaining exercise to work through at your leisure.