Programming Languages: An Active Learning Approach

Chapter 8
Formal Semantics

Kent D. Lee

Formal Semantics

- Language designers have long struggled to find the best way to describe the meaning of a language.
- Most language descriptions rely on English or some other informal language to describe their meaning.
- This area of computer science is referred to as *Formal Semantics of Language Description* or sometimes just *Formal Semantic Methods*.

Why a Formal Method?

Example 8.1

Consider this example of C++ code.

```
int x = 1;
cout << x++ << " " << x << endl;
```

Why a Formal Method?

- You might expect this code to print “1 2” to the screen.
- However, the English description of the C++ language, called the Annotated C++ Reference Manual, doesn’t specify in which order expressions are evaluated within one statement.
- When this code was compiled with two different compilers, the GNU g++ compiled program printed “1 2” while a MIPS C++ compiled version of the program printed “2 1”.

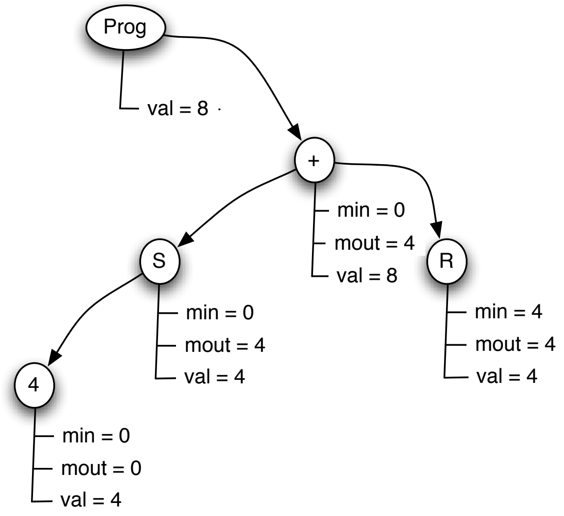
Why a Formal Method?

Example 8.2

This is a very simple Pascal program involving two records that have the same type of fields but different names.

```
program types;
  type A = record
             g:integer;
             h:string;
          end;
       B = record
             i:integer;
             j:string;
           end;
  var x : A;
      y : B;
  begin
    x.g := 5;
    x.h := "hi";
    y := x
  end.
```

Annotated AST for + S 4 R



Attribute Grammars

Example 8.3

Here is the grammar for prefix calculator expressions. The grammar represents prefix expressions because the operation is written before its arguments. So, `+ 5 * 6 4` results in 29 when evaluated. Notice that when written in prefix notation, the expression `S 5` stores 5 in the memory location. `S` is now a prefix operator and not a postfix operator as it was previously defined.

$$
\begin{array}{l}
\text{Prog} \rightarrow \text{Expr EOF} \\
\text{Expr} \rightarrow \text{op Expr Expr} \mid \text{S Expr} \mid \text{number} \mid \text{R} \\
\quad \text{where op is one of } +, -, *, /
\end{array}
$$

The Prog production in the abstract syntax below was added to assist in the definition of the attribute grammar. Notice that parenthesis have disappeared in the concrete syntax as well as practically all the nonterminals. They aren't needed in a prefix expression grammar. The precedence of operators is determined by the order of operations within the expression. The concrete syntax of the language shown above leads to the abstract syntax description below. Since the precedence of operations is determined by the order they appear, the concrete and abstract syntax are nearly identical.

AST $\rightarrow$ Prog AST $|$ op AST AST $|$ Store AST $|$ Recall $|$ number
where op is one of $+, -, *, /$

Attribute Grammars

Example 8.4

Here is the attribute grammar of calculator expressions.

$$
\begin{array}{ll}
\text{AST} & \rightarrow \text{Prog AST} \\
(1) & \text{AST}_1.\text{min} = 0 \\
(2) & \text{AST}_0.\text{val} = \text{AST}_1.\text{val} \\
\text{AST} & \rightarrow \text{op AST AST} \\
(3) & \text{AST}_1.\text{min} = \text{AST}_0.\text{min} \\
(4) & \text{AST}_2.\text{min} = \text{AST}_1.\text{mout} \\
(5) & \text{AST}_0.\text{mout} = \text{AST}_2.\text{mout} \\
(6) & \text{AST}_0.\text{val} = \text{AST}_1.\text{val op AST}_2.\text{val} \\
 & \text{where op is one of } +, -, *, /
\end{array}
$$

$$\begin{array}{ll}
\text{AST} & \rightarrow \text{Store AST} \\
(7) & \text{AST}_1.\text{min} = \text{AST}_0.\text{min} \\
(8) & \text{AST}_0.\text{mout} = \text{AST}_1.\text{val} \\
(9) & \text{AST}_0.\text{val} = \text{AST}_1.\text{val} \\
\text{AST} & \rightarrow \text{Recall} \\
(10) & \text{AST}_0.\text{val} = \text{AST}_0.\text{min} \\
(11) & \text{AST}_0.\text{mout} = \text{AST}_0.\text{min} \\
\text{AST} & \rightarrow \text{number} \\
(12) & \text{AST}_0.\text{mout} = \text{AST}_0.\text{min} \\
(13) & \text{AST}_0.\text{val} = \text{number}
\end{array}$$

Attribute Grammars

☞ Practice 8.1

Justify the annotation of the tree given in figure 1 by stating which rule was used in assigning each of the attributes annotating the tree.

▸ View Solution

Attribute Types

☞ Practice 8.2

Is the min attribute synthesized or inherited? Is the mout attribute synthesized or inherited?

▸ View Solution

Axiomatic Semantics

$$\{P_E^V\} V := E\{P\} \tag{1}$$

$$\frac{\{P\}S_1\{Q\} \text{ and } \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}} \tag{2}$$

$$\frac{P \supset Q \text{ and } \{Q\}S\{R\}}{\{P\}S\{R\}} \tag{3}$$

$$\frac{\{P\}S\{Q\} \text{ and } Q \supset R}{\{P\}S\{R\}} \tag{4}$$

$$\frac{\{P \wedge B\}S_1\{Q\} \text{ and } \{P \wedge B\}S_2\{Q\}}{\{P\} if\ B\ then\ S_1\ else\ S_2\{Q\}} \tag{5}$$

$$\frac{\{P \wedge B\}S\{Q\} \text{ and } P \wedge B \supset Q}{\{P\} if\ B\ then\ S\{Q\}} \tag{6}$$

Axiomatic Semantics

Example 8.5

Consider the following Pascal code to find the maximum of three numbers:

```
m:= i;                  //statement S1
if m < j then m:=j;     //statement S2
if m < k then m:=k;     //statement S3
```

Assume we would like to prove that m is equal to the maximum of i, j, and k (i.e. $m \geq i$ and $m \geq j$ and $m \geq k$) after the sequence of statements above is executed. To prove this we will need to use the rules of inference that have been defined and the assignment axiom. The assignment axiom is easier to use if used backwards. Starting with the last statement, S3, the proof begins by working through the statements in reverse order.

```
{ m<k and m>=i and m>=j }
      implies { k>=i and k>=j and k=k }
m:=k;
{ m>=i and m>=j and m=k }
      implies { m>=i and m>=j and m>=k }
```

This was derived by applying inference rules 4, 2, and 3 in that order by working backward through the assignment statement. The precondition of the assignment statement in S3 is now `{ m<k and m>=i and m>=j }`. If the condition in S3 is false (i.e. `m>=k`) then we automatically have

```
    { m>=i and m>=j and m>=k }
```

which is the post condition of the whole if-then statement. Therefore, according to the If-Then rule, inference rule 1, we have

```
{ m>=i and m>=j } if m < k then m:=k;
      { m>=i and m>=j and m>=k }
```

Similarily, for S2 if `m<j` we find that

```
{ m<j and m=i } implies { j>=i and j=j }
m:=j;
{ m>=i and m=j } implies { m>=i and m>=j }
```

by rules 4, 3, and 2. In S2, if `m>=j` we get

```
{ m=i and m>=j } implies { m>=i and m>=j }
```

Finally, to finish the proof

```
{ true } implies { i=i } m:=i { m=i }
```

Putting it all together results in the following proof.

```
{ true } implies { i=i and j=j and k=k }
m:=i;
{ m=i and j=j and k=k } implies { m>=i and j=j and k=k }
if m < j then m:= j;
{ m>=i and m>=j and k=k }
if m < k then m := k;
{ m>=i and m>=j and m>=k }
```

Action Semantics

- First and foremost, Mosses and Watt wanted a formal language for specifying programming language semantics that was accessible and useful to a wide range of computer scientists, from programmers to language designers.
- Both denotational semantics and action semantics rely on semantic functions to compositionally map an *abstract syntax tree* representing a program to its semantic equivalent, either a *denotation* or an *action*, respectively.

The Small Language

- Variables with assignment
- Iteration - while loops
- Selection - if then else statements
- Functions with zero or more parameters
- Input of ints
- Output of ints and bools

The Small Language

Example 8.6

Here is an example Small program which computes the factorial of a number entered at the keyboard and then prints the result to the screen.

```
let fun fact(x) =
  if x=0 then 1
  else
    (output(x);
     x*fact(x-1))
in
  output(fact(input()))
end
```

Running the program produces the following interaction.

```
? 5
5
4
3
2
1
120
```

The Action of Factorial

```
||bind "output" to native abstraction of an action
||[ using the given (integer|truth-value) ][ giving () ]
|before
||bind "input" to native abstraction of an action
||[ using the given () ][ giving an integer ]
hence
||furthermore
|||recursively
||||bind "fact" to closure of the abstraction of
||||||furthermore
|||||||bind "x" to the given (integer|truth-value)\#1
|||||thence
||||||||||give (integer|truth-value) bound to "x"
|||||||||or
||||||||||give [(integer|truth-value)]cell bound to "x"
||||||||and then
|||||||||give 0
|||||||then
||||||||give the given ((integer|truth-value)|
||||||||                [(integer|truth-value)]cell)\#1
||||||||is the given ((integer|truth-value)|
||||||||                [(integer|truth-value)]cell)\#2
||||||then
```

```
||||||||give 1
|||||||else
|||||||||||give (integer|truth-value) bound to "x"
||||||||||or
|||||||||||give [(integer|truth-value)]cell bound to "x"
|||||||||then
||||||||||enact application of the abstraction of an action
||||||||||bound to "output" to the given data
||||||||then
||||||||||||give (integer|truth-value) bound to "x"
|||||||||||or
||||||||||||give [(integer|truth-value)]cell bound to "x"
||||||||||and then
|||||||||||||||give (integer|truth-value) bound to "x"
||||||||||||||or
|||||||||||||||give [(integer|truth-value)]cell bound to "x"
|||||||||||||and then
||||||||||||||give 1
||||||||||||then
|||||||||||||give the difference of
|||||||||||||(the given integer\#1, the given integer\#2)
|||||||||||then
||||||||||||enact application of the abstraction of an action
||||||||||||bound to "fact" to the given data
|||||||||then
||||||||||give the product of the given data
```

```
|hence
|||||complete
||||then
|||||enact application of the abstraction of an action bound to
|||||"input" to the given data
|||then
||||enact application of the abstraction of an action bound to
||||"fact" to the given data
||then
|||enact application of the abstraction of an action bound to
|||"output" to the given data
```

Data and Sorts

- Several *sorts* of data are available in action semantics, including integers, lists, maps, characters, and strings among others.
- There are two operators in action semantics for constructing sorts from other sorts, the join and meet operators.

  integer **|** truth-value = {false,true,0,-1,1,-2,2,...}
  integer & truth-value = nothing
  false **|** true = truth-value

  s & nothing = nothing
  $s \mid$ nothing = s

The Current Information

- Transients
  Transients represent intermediate, or short-lived, values that are given when evaluating actions. Transients are represented as data, a tuple of datum. For instance, the primitive action ‘give 10’ produces the transient tuple (10).
- Bindings
  Bindings, represented by a finite mapping sort called map, record the binding of identifiers to a sort called bindable data. bindable data, called denotable values in denotational semantics, usually includes cells, values, and abstractions representing procedures and functions in programming languages. Bindings are created through the use of primitive actions like ‘bind “n” to the given value’, which produces a map of the identifier “n” to the bindable value.

- Storage
  Storage represents stable information and is a map of locations, called cells to storable data. Storage is persistent: Once the contents of a cell have been altered, it retains that value until it is changed again. Storage, like bindings, is modified by primitive actions. For instance, the primitive action ‘store the given value#2 in the given variable#1’ stores the datum 10 in the cell bound to the identifier “m” given the appropriate transients and bindings. Note that the sort variable is just another name for the sort cell in this example.

Yielders

- given S
  Assumes that the current transient consists of datum d, which is a subsort of S, and yields d. If d is not a subsort of S, then the yielder yields nothing.
- given S#n
  Assumes that the current transient tuple includes a datum d, a subsort of S, at position n in the tuple. It yields d, if d is a subsort of S and nothing otherwise.
- S bound to id
  Expects that the current bindings includes a mapping of id to d, a subsort of S. If it does, then the yielder yields d. If d is not a subsort of S or there is no binding of id in the bindings, then it yields nothing.

- S stored in C
  Assumes that the current storage maps location C to a storable datum d, where d is a subsort of S, and yields d. Otherwise, it yields nothing.

Primitive Actions

Example 8.7

Here are some examples of primitive actions.

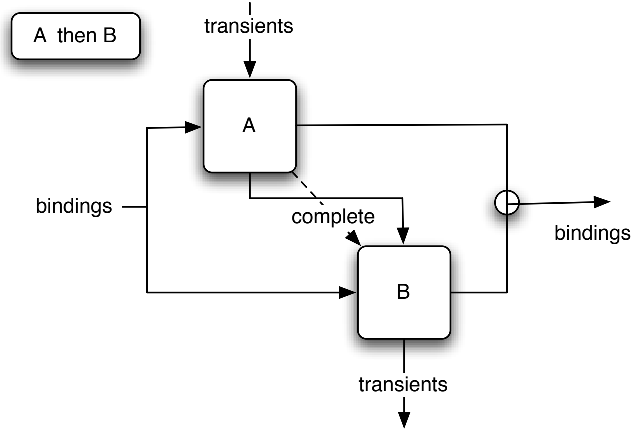
- allocate a cell
    - sets aside a new cell in storage and gives it as a transient
- bind M to 5
    - produces the new binding in the outgoing bindings
- store 6 in the given cell
    - stores 6 by mapping the given cell to the value 6.

Combinators and Facets

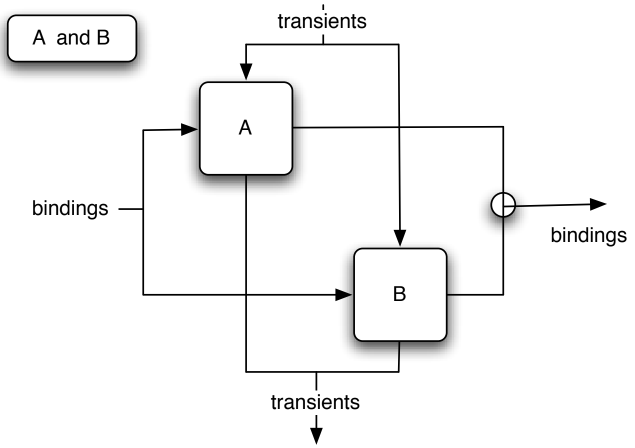
- Action semantics would not be very interesting if primitive actions could not be combined to produce more complex actions.
- *Combinators* are used to construct *compound actions* from *subactions*.
- Examples of combinators are and, then, hence, and furthermore.
- Combinators are defined to propagate parts of the current information and are also responsible for controlling the order of performance of their subactions.

```
| give 10
then
| bind “n” to the given value
```

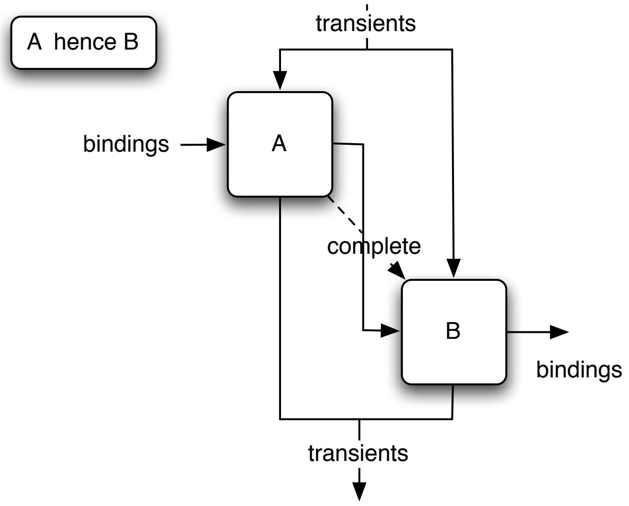
The then Diagram



The and Diagram



The hence Diagram



Incomes and Outcomes

- It is possible to restrict actions to a subsort of action by the use of incomes and outcomes.

  give the given data [using a given integer]

  give the given data [giving an integer]

  bind “m” to the given integer [binding]

  give the given data [giving an integer] [using an integer] [completing]

  sum (the given integer#1, the given integer#2) [an integer]

Action Semantic Descriptions

- An Action Semantic Description maps the syntax of a programming language to its action semantics.
- The mapping is given by a set of semantic functions and equations.

Semantic Functions and Equations

- A semantic function is a function from a syntactic category to an action.
    - evaluate _ :: Expr $\rightarrow$ action

(7) evaluate ⟦ “if” S:SExpr “then” E_1:Expr “else” E_2:Expr ⟧
=

```
        | evaluateSExpr $S$
        then
        | | evaluate $E_1$
        | else
        | | evaluate $E_2$
```

Exercises

1. Using the attribute grammar, construct a decorated abstract syntax tree for the expression + * S 6 5 R. Justify the assignment of attributes by referring to each rule that governed the assignment of a value in the annotated tree.
2. In this exercise you are to construct an interpreter for prefix expressions with the addition of a single memory location (like the interpreter you previously constructed, but implemented in Prolog this time). Your grammar should contain parameters that build an abstract syntax tree of the expression. Then, you should write Prolog rules that evaluate the abstract syntax tree to get the resulting expression. The grammar for prefix expressions is given in example 8.3. The grammar is LL(1) so no modifications of it are necessary to generate a parser for it in Prolog.

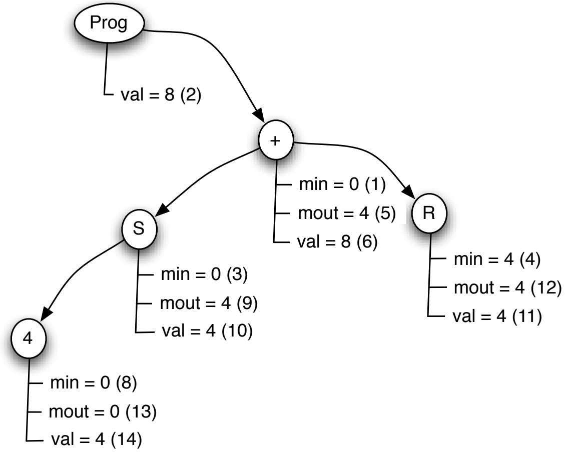
To complete this project you will want to use the readln predicate described in section **??**. However, to make things easier while parsing, you should preprocess the list so that an expression like “+ S 5 R”, which readln returns as [+,s,5,r], will look like [+,s,num(5),r] after preprocessing. The num structure for numbers will help you when you write the parser.

HINT: This assignment is very closely related to the attribute grammar given in example 8.4. The main predicate for the interpreter should approximate this:

```
calc :- read a line, preprocess the line, parse the expression,
        evaluate the AST, print the result.
```

3. Download the Genesis compiler generator and use it to generate the Small compiler. Then use the compiler to compile a Small program. The Small Action Semantic Description is provided with the downloadable Genesis compiler generator. Complete instructions for using Genesis are available on the text's web page.
4. Using Genesis, write an action semantics for the simple calculator expression language. You may use either the prefix version of the grammar given in this chapter or the infix version of the grammar presented in previous chapters. Create the semantic functions that map the concrete syntax of the language directly to their actions. Test your creation to be sure it works. Remember, Genesis is a research project and no guarantees are made regarding appropriate error messages. However, it should be pretty stable as it is written in SML.

Solution to Practice Problem 8.1



Go Back to Practice Problem

Solution to Practice Problem 8.2

The val attribute is synthesized. The min value is inherited. The mout value is synthesized.

▸ Go Back to Practice Problem