*12*

## Building Interpreters

*Interpreters are programs that execute DSL programs by directly traversing the DSL program and performing the semantic actions associated with the respective program elements. The chapter contains examples for interpreters with Xtext, MPS and Spoofax.*

|  |  |
| --- | --- |
| How an interpreter implementation looks like depends a lot on the programming language used for implementing it. Also, the complexity of the interpreter directly reflects the complexity of the language it processes in terms of size, structure and semantics2. The following list explains some typical ingredients |  |
| that go into building interpreters for functional and procedural languages. It assumes a programming language that can polymorphically invoke functions or methods.  *Expressions* For program elements that can be evaluated to values, i.e., expressions, there is typically a function **eval** that is defined for the various expression concepts in the language, i.e. it is polymorphically overridden for subconcepts of **Expression**. Since nested expressions are almost always represented as nested trees in the AST, the **eval** function calls it- |  |
| self with the program elements it owns, and then performs some semantic action on the result3. Consider an expression |  |

Interpreters are programs that read a model, traverse the AST and perform actions corresponding to the execution semantics of the language constructs whose instances appear in the AST1.

**3 \* 2 + 5**. Since the **+** is at the root of the AST, **eval(Plus)** would be called (by some outside entity). It is implemented to add the values obtained by evaluating its arguments. So it

same structure: the **eval** template for some kind of expression calls the **eval** templates for its children.

calls **eval(Multi)** and **eval(5)**. Evaluating a number literal is trivial, since it simply returns the number itself. **eval(Multi)** would call **eval(3)** and **eval(2)**, multiplying their results and returning the result of the multiplication as its own result, allowing plus to finish its calculation.

|  |  |
| --- | --- |
|  |  |
| *Statements* Program elements that don’t produce a value only make sense in programming languages that have side effects. In other words, execution of such a language concept produces some effect either on global data in the program (reassignable variables, object state) or on the environment of the program (sending network data or rendering a UI). Such program elements are typically called **Statements**. Statements are either arranged in a list (typically called a statement list) or arranged recursively nested as a tree (an **if** statement has a **then** and an **else** part which are themselves statements or statement lists). To execute those, there is typically a function **execute** that is overloaded for all of the different state- |  |
| ment types4. Note that statements often contain expressions |  |
| and more statement lists (as in **if (a > 3) { print a; a=0; } else { a=1;}**), so an implementation of **execute** may call **eval** and perform some action based on the result (such as deciding whether to execute the **then**-part of the **else**-part of the **if** statement). Executing the **then**-part and the **else**-part boils down to calling **execute** on the respective statement lists.  *Environments* Languages that support assignment to variables (or modify any other global state) require an environment during execution to remember the values for the variables at |  |
| each point during program execution5. The interpreter must |  |
| keep some kind of global hash table, known as the *environment*, to keep track of symbols and their values, so it can look them up when evaluating a reference to that symbol. Many (though not all) languages that support assignable variables allow re- |  |

assignment to the same variable (as we do in **a = a + 1;**). In this case, the environment must be updateable. Notice that in **a = a + 1** both mentions of **a** are references to the same variable, and both **a** and **a+1** are expressions. However, only **a** (and not **a + 1**) can be assigned to: writing **a + 1 = 10 \* a;** would be invalid. The notion of an **lvalue** is introduced to describe this. lvalues can be used "on the left side" of an assignment. Variable references are typically lvalues (if they don’t point to a **const** variable). Complex expressions usually are not[[1]](#footnote-1).

*Call Stacks* The ability to call other entities (functions, procedures, methods) introduces further complexity, especially regarding parameter and return value passing, and the values of local variables. Assume the following function:

**int** add(**int** a, **int** b) { **return** a + b; }

When this function is called via **add(2, 3)** the actual arguments **2** and **3** have to be bound to the formal arguments **a** and **b**. An environment must be established for the execution of **add** that keeps track of these assignments7. Now consider the

7

|  |  |
| --- | --- |
| following recursive function: |  |

environment must delegate to the **int** fac(**int** i) {

**return** i == 0 ? 1 : fac(i - 1); global environment in case a referenced

} symbol cannot be found in the local

|  |  |
| --- | --- |
| In this case, each recursive call to **fac** requires that a new environment is created, with its own binding for the formal variables. However, the original environment must be "remembered" because it is needed to complete the execution of the outer **fac** after a recursively called **fac** returns. This is achieved using a stack of environments. A new environment is pushed onto the stack as a function is called (recursively), and the stack is popped, returning to the previous environment, as a called function returns. The return value, which is often expressed using some kind of **return** statement, is usually placed into the inner environment using a special symbol or name (such as **\_\_ret\_\_**). It can then be picked up from there as the inner environment is popped. | environment. |
| *12.1 Building an Interpreter with Xtext* |  |

This example describes an interpreter for the cooling language[[2]](#footnote-2). It is used to allow DSL users to "play" with the cooling programs before or instead of generating C code. The interpreter can execute test cases (and report success or failure) as well as simulate the program interactively. Since no code generation and no real target hardware is involved, the turn-around time is much shorter and the required infrastructure is trivial – only the IDE is needed to run the interpreter. The execution engine, as the interpreter is called here, has to handle the following language aspects:

cussed earlier: **code.google.com/a/ eclipselabs.org/p/ xtext-typesystem/**

* The DSL supports expressions and statements, for example in the entry and exit actions of states. These have to be supported in the way described above.
* The top-level structure of a cooling program is a state machine. So the interpreter has to deal with states, events and transitions.
* The language supports deferred execution (i.e. perform a set of statements at a later time), so the interpreter has to keep track of deferred parts of the program.
* The language supports writing tests for cooling programs, including mock behavior for hardware elements. A set of constructs exists to express this mock behavior (specifically, ramps to change temperatures over time). These background tasks must be handled by the interpreter as well.

*Expressions and Statements* We start our description of the execution engine inside out, by looking at the interpreter for expressions and statements first. As mentioned above, for interpreting expressions, there is typically an overloaded **eval** operation, that contains the implementation of expression evaluation for each subtype of a generic **Expression** concept. However, Java doesn’t have polymorphically overloaded member methods9. We compensate this by generating a dispatcher

|  |  |
| --- | --- |
| that calls a *different* method for each subtype of **Expression**10. |  |
| The generation of this dispatcher is integrated with Xtext via a workflow fragment, i.e. the dispatcher is generated during the overall Xtext code generation process. The fragment is configured with the abstract meta classes for expressions and statements. The following code shows the fragment configuration: |  |

**fragment** = de.itemis.interpreter.generator.InterpreterGenerator { interpreter would have been similar. expressionRootClassName = "Expression"

statementRootClassName = "Statement"

}

This fragment generates an abstract class that acts as the basis for the interpreter for the particular set of statements and expressions. As the following piece of code shows, the expression evaluator class contains an **eval** method that uses **instanceof** checks to dispatch to a method specific to the subclass, thereby emulating polymorphically overloaded methods[[3]](#footnote-3). The specific methods throw an exception and are expected to be overridden by a manually written subclass that contains the actual interpreter logic for the particular language concepts[[4]](#footnote-4):

|  |
| --- |
| **public abstract class** AbstractCoolingLanguageExpressionEvaluator **extends** AbstractExpressionEvaluator {  **public** AbstractCoolingLanguageExpressionEvaluator(ExecutionContext ctx) { **super**(ctx);  }  **public** Object eval( EObject expr, LogEntry parentLog ) **throws** InterpreterException {  LogEntry localLog = parentLog.child(LogEntry.Kind.eval, expr,  "evaluating "+expr.eClass().getName());  **if** ( expr **instanceof** Equals ) **return** evalEquals( (Equals)expr, localLog );  **if** ( expr **instanceof** Unequals ) **return** evalUnequals( (Unequals)expr, localLog );  **if** ( expr **instanceof** Greater ) **return** evalGreater( (Greater)expr, localLog ); // the others...  }  **protected** Object evalEquals( Equals expr, LogEntry log ) **throws** InterpreterException {  **throw new** MethodNotImplementedException(expr,  "evalEquals not implemented");  }  **protected** Object evalUnequals( Unequals expr, LogEntry log )  **throws** InterpreterException {  **throw new** MethodNotImplementedException(expr,  "evalUnequals not implemented");  }  // the others... } |

Before we dive into the details of the interpreter code below, it is worth mentioning that the "global data" held by the execution engine is stored and passed around using an instance of

**EngineExecutionContext**. For example, it contains the environment that keeps track of symbol values, and it also has access to the type system implementation class for the language. The **ExecutionContext** is available through the **eec()** method in the **StatementExecutor** and **ExpressionEvaluator**.

Let us now look at some example method implementations. The following code shows the implementation of **evalNumberLiteral**, which evaluates number literals such as **2** or **2.3** or **-10.2**. To recap, the following grammar is used for defining number literals:

|  |
| --- |
| Atomic **returns** Expression:  ...  ({NumberLiteral} value=DECIMAL\_NUMBER);  **terminal** DECIMAL\_NUMBER:  ("-")? (’0’..’9’)\* (’.’ (’0’..’9’)+)?; |

With this in mind, the implementation of **evalNumberLiteral** should be easily understandable. We first retrieve the actual value from the **NumberLiteral** object, and we find the type of the number literal using the **typeof** function in the type sys-

|  |  |  |  |
| --- | --- | --- | --- |
| tem13. Based on this distinction, **evalNumberLiteral** returns | | |  |
| either a Java **Double** or **Integer** as the value of the **NumberLiteral**. In addition, it creates log entries that document these decisions. | | |  |
| **protected** Object evalNumberLiteral(NumberLiteral expr, LogEntry log) {  String v = ((NumberLiteral) expr).getValue();  EObject type = eec().typesystem.typeof(expr,  **new** TypeCalculationTrace());  **if** (type **instanceof** DoubleType) { log.child(Kind.debug, expr, "value is a double, " + v); **return** Double.valueOf(v);  } **else if** (type **instanceof** IntType) { log.child(Kind.debug, expr, "value is a int, " + v); **return** Integer.valueOf(v);  }  **return null**;  } |

The evaluator for **NumberLiteral** was simple because number literals are leaves in the AST and have no children, and no recursive invocations of **eval** are required. This is different for the **LogicalAnd**, which has two children in the **left** and **right** properties. The following code shows the implementation of **evalLogicalAnd**.

|  |
| --- |
| **protected** Object evalLogicalAnd(LogicalAnd expr, LogEntry log) { **boolean** leftVal = ((Boolean)evalCheckNullLog( expr.getLeft(), log ))  .booleanValue();  **if** ( !leftVal ) **return false**; **boolean** rightVal = ((Boolean)evalCheckNullLog( expr.getRight(), log ))  .booleanValue();  **return** rightVal;  } |
| The first statement calls the evaluator, for the **left** property.  If **leftVal** is **false** we return without evaluating the right ar- | | |  |
| gument14. If it is true we evaluate the right argument[[5]](#footnote-5). The | | | 14 Most programming languages never |

value of the **LogicalAnd** is then the value of **rightVal**.

So far, we haven’t used the environment, since we haven’t worked with variables and their (changing) values. Let’s now look at how variable assignment is handled. We first look at the **AssignmentStatement**, which is implemented in the **StatementExecutor**:

|  |
| --- |
| **protected void** executeAssignmentStatement( AssignmentStatement s,  LogEntry log) {  Object l = s.getLeft();  Object r = evalCheckNullLog(s.getRight(), log);  SymbolRef sr = (SymbolRef) l;  SymbolDeclaration symbol = sr.getSymbol(); eec().environment.put(symbol, r);  log.child(Kind.debug, s, "setting " + symbol.getName() + " to " + r);  } |

The first two lines get the **left** argument as well as the value of the **right** argument. Note how only the right value is evaluated: the left argument is a symbol reference (ensured through

a constraint, since only **SymbolRef**s are lvalues in this language). We then retrieve the symbol referenced by the symbol reference and create a mapping from the symbol to the value in the environment, effectively "assigning" the value to the symbol during the execution of the interpreter.

The implementation of the **ExpressionEvaluator** for a symbol reference (if it is used not as an lvalue) is shown in the following code. We use the same environment to look up the value for the symbol. We then check whether the value is **null** (i.e. nothing has been assigned to the symbol as yet). In this case we return the default value for the respective type and log

a warning16, otherwise we return the value.

|  |
| --- |
| **protected** Object evalSymbolRef(SymbolRef expr, LogEntry log) {  SymbolDeclaration s = expr.getSymbol(); Object val = eec().environment.get(s); **if** (val == **null**) {  EObject type = eec().typesystem.typeof(expr,  **new** TypeCalculationTrace()); Object neutral = intDoubleNeutralValue(type); log.child(Kind.debug, expr,  "looking up value; nothing found, using neutral value: " + neutral);  **return** neutral;  } **else** { log.child(Kind.debug, expr, "looking up value: " + val); **return** val;  }  } |

|  |  |
| --- | --- |
| The cooling language does not support function calls, so we demonstrate function calls with a similar language that supports them. In that language, function calls are expressed as symbol references that have argument lists. Constraints make sure that argument lists are only used if the referenced symbol |  |
| is actually a **FunctionDeclaration**17. Here is the grammar. |  |

|  |
| --- |
| FunctionDeclaration **returns** Symbol:  {FunctionDeclaration} "function" type=Type name=**ID** "("  (params+=Parameter ("," params+=Parameter)\* )? ")" "{"  (statements+=Statement)\*  "}";  Atomic **returns** Expression:  ...  {SymbolRef} symbol=[Symbol|QID]  ("(" (actuals+=Expr)? ("," actuals+=Expr)\* ")")?; |

|  |  |
| --- | --- |
| The following is the code for the evaluation function for the symbol reference. It must distinguish between references to |  |
| variables and to functions18. |  |

|  |
| --- |
| **protected** Object evalSymbolRef(SymbolRef expr, LogEntry log) { Symbol symbol = expr.getSymbol(); **if** ( symbol **instanceof** VarDecl ) { **return** log( symbol, eec().environment.getCheckNull(symbol, log) );  }  **if** ( symbol **instanceof** FunctionDeclaration ) {  FunctionDeclaration fd = (FunctionDeclaration) symbol; |

|  |
| --- |
| **return** callAndReturnWithPositionalArgs("calling "+fd.getName(),  fd.getParams(), expr.getActuals(), fd.getElements(), RETURN\_SYMBOL, log);  } **throw new** InterpreterException(expr,  "interpreter failed; cannot resolve symbol reference " +expr.eClass().getName()); } |

The code for handling the **FunctionDeclaration** uses a prede-

fined utility method **callAndReturnWithPositionalArgs**19. the list of formal arguments of the called

function, the list of actual arguments (expressions) passed in at the call site, the list of statements in the function body, a symbol that should be used for the return value in the environment, as well as the obligatory log. The utility method is implemented as follows:

|  |
| --- |
| **protected** Object callAndReturnWithPositionalArgs(String name,  EList<? **extends** EObject> formals, EList<? **extends** EObject> actuals,  EList<? **extends** EObject> bodyStatements) { eec().environment.push(name); **for**( **int** i=0; i<actuals.size(); i++ ) {  EObject actual = actuals.get(i); EObject formal = formals.get(i); eec().environment.put(formal, evalCheckNullLog(actual, log));  }  eec().getExecutor().execute( bodyStatements, log ); Object res = eec().environment.get(RETURN\_SYMBOL); eec().environment.pop(); **return** res;  } |

Remember that each invocation of a function has to get its own environment to handle the local variables for the particular invocation. We can see this in the first line of the implementation above: we first create a new environment and push it onto the call stack. Then the implementation iterates over the actual arguments, evaluates each of them and "assigns" them to the formals by creating an association between the formal argument symbol and the actual argument value in the new environment. It then uses the **StatementExecutor** to execute all the statements in the body of the function. Notice that as the executed function deals with its own variables and function calls, it uses the *new* environment created, pushed onto the stack and populated by this method. When the execution of the body has finished, we retrieve the return value from the environment. The **return** statement in the function has put it there under a name we have prescribed, the **RETURN\_SYMBOL**, so we know how to find it in the environment. Finally, we pop the environment, restoring the caller’s state of the world and return the return value as the resulting value of the function call expression.

*States, Events and the Main program* Changing a state20 from within a cooling program is done by executing a **ChangeStateStatement**, which simply references the state that should be entered. Here is the interpreter code in **StatementExecutor**:

|  |
| --- |
| **protected void** executeChangeStateStatement(ChangeStateStatement s,  LogEntry l) {  engine.enterState(s.getTargetState(), log);  }  **public void** enterState(State targetState, LogEntry logger ) **throws** TestFailedException, InterpreterException,  TestStoppedException {  logger.child( Kind.info, targetState,  "entering state "+targetState.getName());  context.currentState = targetState;  executor.execute(ss.getEntryStatements(), logger); **throw new** NewStateEntered(); } |

**executeChangeStateStatement** calls back to an engine method that handles the state change21. The method sets the current

1. State as in state machine, not as in program state.

state to the target state passed into the method (the current state is kept track of in the execution context). It then executes the set of entry statements of the new state. After this it throws an exception **NewStateEntered**, which stops the current execution step. The overall engine is step driven, i.e. an external "timer" triggers distinct execution steps of the engine. A state change always terminates the current step. The main method **step()** triggered by the external timer can be considered the main program of the interpreter. It looks like this:

|  |
| --- |
| **public int** step(LogEntry logger) { **try** {  context.currentStep++; executor.execute(getCurrentState().getEachTimeStatements(), stepLogger);  executeAsyncStuff(logger);  **if** ( !context.eventQueue.isEmpty() ) {  CustomEvent event = context.eventQueue.remove(0);  LogEntry evLog = logger.child(Kind.info, **null**,  "processing event from queue: "+event.getName()); processEventFromQueue( event, evLog ); **return** context.currentStep;  }  processSignalHandlers(stepLogger); } **catch** ( NewStateEntered ignore ) {} **return** context.currentStep; } |

It first increments a counter that keeps track of how many steps have been executed since the interpreter has been started. It then executes the **each time** statements of the current state. This is a statement list defined by a state that needs to be reexecuted in each step while the system is in the respective state. It then executes asynchronous tasks. We’ll explain those below. Next it checks if an event is in the event queue. If so, it removes the first event from the queue and executes it. After processing

an event the step is always terminated. Lastly, if there was no event to be processed, we process signal handlers (the **check** statements in the cooling programs).

Processing events checks whether the current state declares an event handler that can deal with the currently processed event. If so, it executes the statement list associated with this event handler.

|  |
| --- |
| **private void** processEventFromQueue(CustomEvent event, LogEntry logger) { **for** ( EventHandler eh: getCurrentState().getEventHandlers()) { **if** ( reactsOn( eh, event ) ) {  executor.execute(eh.getStatements(), logger); }  }  } |

The DSL also supports executing code asynchronously, i.e. after a specified number of steps (representing logical program time). The grammar looks as follows:

|  |
| --- |
| PerformAsyncStatement:  "perform" "after" time=Expr "{" (statements+=Statement)\* "}"; |

The following method interprets the **PerformAsyncStatement**s:

**protected void** executePerformAsyncStatement(PerformAsyncStatement s,

LogEntry log) **throws** InterpreterException { **int** inSteps = ((Integer)evalCheckNullLog(s.getTime(), log)).intValue(); eec().asyncElements.add(**new** AsyncPerform(eec().currentStep + inSteps,

"perform async", s, s.getStatements()));

}

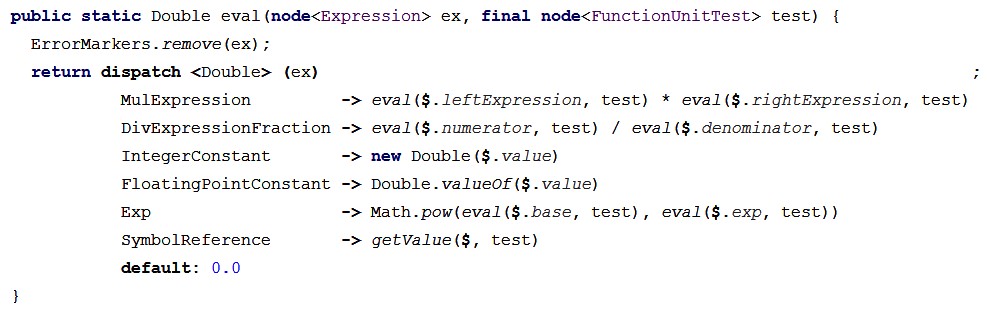
It registers the statement list associated with the **PerformAsyncStatement** in the list of async elements in the execution context. The call to **executeAsyncStuff** at the beginning of the **step** method described above checks whether the time has come and executes those statements:

|  |
| --- |
| **private void** executeAsyncStuff(LogEntry logger) {  List<AsyncElement> stuffToRun = **new** ArrayList<AsyncElement>(); **for** (AsyncElement e: context.asyncElements) { **if** ( e.executeNow(context.currentStep) ) { stuffToRun.add(e);  } } **for** (AsyncElement e : stuffToRun) { context.asyncElements.remove(e);  e.execute(context, logger.child(Kind.info, **null**, "Async "+e));  }  } |

### 12.2 An Interpreter in MPS

Building an interpreter in MPS is essentially similar to building an interpreter in Xtext and EMF. All concepts would apply in

|  |  |
| --- | --- |
| the same way22. However, since MPS’ BaseLanguage is itself |  |
| built with MPS, it can be extended. So instead of using a generator to generate the dispatcher that calls the **eval** methods for the expression classes, suitable modular language extensions can be defined in the first place. | with the **node<>** types that are available on MPS to access ASTs. |
| *A Dispatch Expression* For example, BaseLanguage could be extended with support for polymorphic dispatch (similar to what Xtend does with **dispatch** methods). An alternative solution involves a dispatch expression, a kind of "pimped switch".  Fig. 12.1 shows an example. |  |

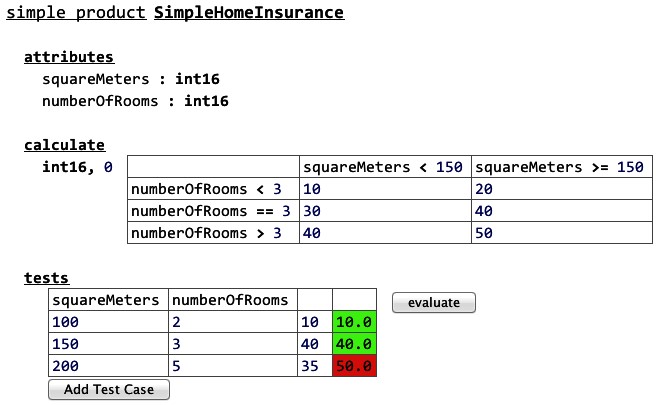


|  |  |
| --- | --- |
| The dispatch expression tests whether the argument **ex** is an instance of the type referenced in the cases. If so, the code on the right side of the arrow is executed. Notice the special expression **$** used on the right side of the arrow. It refers to the |  |
| argument **ex**, but it is already downcast to the type on the left of the case’s arrow. This way, writing annoying downcasts for each property access can be avoided.  Note that this extension is modular in the sense that the definition of BaseLanguage was not changed. Instead, an additional language module was defined that *extends* BaseLanguage. This module can be used as part of the program that contains the interpreter, making the **dispatch** statement available there23. Also, the **$** expression is restricted to only be usable on the right side of the **->**, allowing the overall base language namespace to be kept clean24. |  |

*Showing Results in the Editor* Since MPS is a projectional editor, it can show things in the editor that are read-only. For example, the result of an interpreter run can be integrated directly into the editor. In Fig. 12.2, the bottom table contains test cases for the **calculate** rule. Users enter a number for

**squareMeters**, **numberOfRooms** and the expected result, and in the last column, the editor shows the actual result of the computation (colored red or green depending on whether the actual result matches the expected result).

Figure 12.2: This screenshot shows an mbeddr-based demo application in which users can specify insurance rules. The system includes an interpreter that executes the test cases directly in the IDE.



The interpreter is integrated via the **evaluate** button25. Its

action listener triggers the computation:

|  |
| --- |
| **component provider**: (editorContext, node)->JComponent { JButton evalButton = **new** JButton("evaluate"); **final** node<ProductTestSuite> suite = node; evalButton.addActionListener(**new** ActionListener() { **public void** actionPerformed(ActionEvent p0) { command {  **foreach** tc **in** suite.cases { **float** result = **new** RateCalculator(tc).calculate( suite.ancestor<ProductType, +>.rateCalculation);  tc.actualResult.value = result + ""; }  }  } });  **return** evalButton;  } |

The coloring of the cells (and making them read-only) is done via query-based formatting:

|  |
| --- |
| **style** { editable : **false** background-color : (node, editorContext)->Color {  node.ancestor<TestCase>.isOK()  ? Color.GREEN : Color.RED;  }  } |

### 12.3 An Interpreter in Spoofax

So far we have seen procedural/functional interpreters in which the program’s execution state is separate from the program it-

components, and these Swing components can react to their own events and modify the model in arbitrary ways.

self. As the interpreter runs, it updates the execution state. Another approach to writing interpreters is state-based interpreters, where the execution of the interpreter is expressed as a set of transformations between program states. State-based interpreters can be specified with rewrite rules in Spoofax, realizing transitions between execution states. This requires:

* A representation of *states*. The simplest way to represent states are terms, but we can also define a new DSL for representing the states in concrete syntax.
* An *initialization transformation* from a program in the DSL to the initial state of the interpreter.
* A *step transformation* from an actual state of the interpreter to the next state of the interpreter.

In the remainder of the section, we develop an interpreter for a subset of Mobl. We start with a simple interpreter for expressions, which we then extend to handle statements.

#### 12.3.1 An Interpreter for Expressions

If we want to evaluate simple arithmetic expressions without variables, the expression itself is the state of the interpreter26.

Thus, no extra term signatures are needed and the initialization transformation is given by identity. For the step transformation, we can define rewrite rules for the different expression kinds:

|  |
| --- |
| eval: Add(Int(x), Int(y)) -> Int(z) **where** z := <add> (x, y) eval: Mul(Int(x), Int(y)) -> Int(z) **where** z := <mul> (x, y)  eval: Not(True()) -> False() eval: Not(False()) -> True()  eval: And(True(), True()) -> True() eval: And(True(), False()) -> False() eval: And(False(), True()) -> False() eval: And(False(), False()) -> False()  eval: LazyAnd(True(), True()) -> True() eval: LazyAnd(False(), \_) -> False() eval: LazyAnd(\_, False(), \_) -> False() |

We can orchestrate these rules in two different styles. First, we can define an interpreter which performs only a single evaluation step by applying one rule in each step:

eval-one: exp -> <oncebu(eval)> exp

Here, **oncebu** tries to apply **eval** at one position in the tree, starting from the leaves (the **bu** in **oncebu** stands for bottomup). We could also use **oncetd**, traversing the tree top-down.

we discussed building debuggers for purely functional languages, and in particular, expression languages. We argued that a debugger is trivial, because there is no real "flow" of the program; instead, the expression can be debugged by simply showing the values of all intermediate expressions in a treelike for. We exploit the same "flowless" nature of pure expression languages when building this interpreter.

However, evaluations are likely to happen at the bottom of the tree, which is why **oncebu** is the better choice. The result of **eval-one** will be a slightly simpler expression, which might need further evaluation. Alternatively, we can directly apply as many rules as possible, trying to evaluate the whole expression:

eval-all: exp -> <bottomup(try(eval))> exp

Here, **bottomup** tries to apply **eval** at every node, starting at the leaves. The result of **eval-all** will be the final result of the expression.

#### 12.3.2 An Interpreter for Statements

If we want to evaluate statements, we need states which capture the value of variables and the list of statements which needs to be evaluated. We can define these states with a signature for terms:

|  |
| --- |
| **signature constructors**  : ID \* IntValue -> VarValue  : ID \* BoolValue -> VarValue  State: List(VarValue) \* List(Statement) -> State |

The first two rules define binary tuples which combine a variable name (**ID**) and a value (either **IntValue** or **BoolValue**). The last rule defines a binary constructor **State**, which combines a list of variable values with a list of statements27. We

first have to adapt the evaluation of expressions to handle variable references in expressions.

eval(|varvals): exp -> <eval> exp eval(|varvals): VarRef(var) -> <lookup> (var, varvals)

eval-one(|s): exp -> <oncebu(eval(|s))> exp eval-all(|s): exp -> <bottomup(try(eval(|s)))> exp

The first two rules take the actual list of variable values of the interpreter (**varvals**) as a parameter. The first rule integrates the existing evaluation rules, which do not need any state information. The second rule looks up the current value **val** of the variable **var** in the list of variable values. The last two rules define small-step and big-step interpreters of expressions, just as before.

We can now define evaluation rules for statements. These rules rewrite the current state into a new state:

|  |
| --- |
| eval:  (varvals, [Declare(var, exp)|stmts]) -> (varvals’, stmts) **where** |

|  |
| --- |
| val := <eval-all(|varvals)> exp; varvals’ := <update> ((var, val), varvals)  eval:  (varvals, [Assign(VarRef(var), exp)|stmts]) -> (varvals’, stmts) **where**  val := <eval-all(|varvals)> exp; varvals’ := <update> ((var, val), varvals)  eval:  (varval, [Block(stmts1)|stmts2]) -> (varvals, <conc> (stmts1, stmts2)) |

The first rule handles variable declarations. On the left-hand side, it matches the current list of variable values **varvals**, the declaration statement **Declare(var, exp)**, and the list of remaining statements **stmts**. It evaluates the expression to a value and updates the list of variable values. The new state (on the right-hand side of the signature) consists of the updated variable values and the list of remaining statements28. The

second rule handling assignments is quite similar. The third rule handles block statements, by concatenating the statements from a block with the remaining statements. The following rule handle an **if** statement:

|  |
| --- |
| eval:  (varvals, [If(exp, thenStmt, elseStmt)|stmts]) -> (varvals, [stmt|stmts]) **where** val := <eval-all(|varvals)> exp; **if** !val => True() **then** stmt := thenStmt  **else**  !val => False(); stmt := elseStmt **end** |

First, it evaluates the condition. Depending on the result, it chooses the next statement to evaluate. When the result is **True()**, the statement from the **thenStmt** branch is chosen. Otherwise the result has to be **False()** and the statement from the **elseStmt** branch is chosen. If the result is neither **True()** nor **False()**, the rule will fail. This ensures that the rule only works when the condition can be evaluated to a Boolean value. The following rule handles **while** loops:

|  |
| --- |
| eval:  (varvals, [While(exp, body)|stmts]) -> (varvals, stmts’) **where** val := <eval-all(|varvals)> exp; **if** !val => True() **then** stmts’ := [body, While(exp, body)|stmts]  **else**  !val => False(); stmts’ := stmts **end** |

Again, the condition is evaluated first. If it evaluates to **True()**, the list of statements is updated to the body of the loop, the **while** loop again, followed by the remaining statements. If it

evaluates to **False()**, only the remaining statements need to be evaluated.

The **eval** rules already define a small-step interpreter, going from one evaluation state to the next. We can define a big-step interpreter by adding a driver, which repeats the evaluation until it reaches a final state:

eval-all: state -> <repeat(eval)> state

#### 12.3.3 More Advanced Interpreters

We can extend the interpreter to handle function calls and objects in a similar way as we did for statements. First, we always have to think about the states of the extended interpreter. Functions will require a call stack, objects will require a heap. Next, we need to consider how the old rules can deal with the new states. Adjustments might be needed. For example, when we support objects, the heap needs to be passed to expressions. Expressions which create objects will change the heap, so we cannot only pass it, but have to propagate the changes back to the caller.

#### 12.3.4 IDE Integration

We can integrate interpreters as builders into the IDE. For bigstep interpreters, we can simply calculate the overall execution result and show it to the user. For small-step interpreters, we can use the initialization transformation in the builder. This will create an initial state for the interpreter. When we define a concrete syntax for these states, they can be shown in an editor. The transition transformation can then be integrated as a refactoring on states, changing the current state to the next one. In this way, the user can control the execution, undo steps, or even modify the current state.

1. [↑](#footnote-ref-1)
2. . [↑](#footnote-ref-2)
3. . [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. . [↑](#footnote-ref-5)