CAS 703 Term Project: Validated General-Purpose Calculators

Hassan Zaker Jason Balaci

McMaster University
{zakerzah, balacij}@mcmaster.ca
April 10, 2023

Contents

Listings

1 Introduction

For our term project in CAS 703 [1], we decided to build a language for describing calculation schemes. The descriptions should be mostly understandable to anyone who has worked with Excel [2] or has used any kind of calculation software. We aim to validate the coherence[1] of the calculator descriptions. Additionally, through generative techniques, we hope to decrease the barrier to entry (as much as we can) of basic software development of calculator programs by defining a transformation of the calculator descriptions to various programming languages[2].

1.1 Objective

We aim to:

1. design a metamodel for describing calculator programs (Section 2),
2. build a concrete syntax for the metamodel, and an Integrated Development Environment (IDE) for said concrete syntax (Section 3),
3. design a set of rules that define "coherence" rules of the metamodel and audit instances of the metamodel for coherence (Section 4), and
4. define a transformer that converts the calculator description into programs and corresponding documentation (Section 5).

1.2 Tooling

We will use the tooling shown in 703, namely: Eclipse Epsilon [3] and the languages it contains, and Xtext [4].

[1] "Coherence" defined by an unambiguous set of constraints and rules.
[2] Notably, Java programs.

2 Modelling

To define our metamodel for our desired Domain-Specific Language (DSL), we first need to iron out the *requirements* of that language.

2.1 Requirements

When we (the authors) think of "calculators," we think of 3 main components: (i) a set of input symbols, (ii) a set of output symbols, and (iii) a means of deriving the output symbols from the input symbols. (i) and (ii) are really just "symbols" with designation of whether they are user-provided or calculated. (iii) is a (commonly secret) algorithm that translates (i) to (ii) and which may rely on intermediate symbols irrelevant to (i) and (ii). Thus, we think of a calculator as being a structure of (a) symbols of different kinds (inputs, intermediates, and outputs) and (b) an algorithm that calculates some symbols from an initial subset of provided symbols.

Depending on your preferred school of thought, algorithms can be thought of differently. We choose to think of them as being similar to imperative-style programming languages, where there is a memory of symbols to values, and a sequence of steps/statements that are executed sequentially against the memory.

The most important kind of "step" our algorithm must have is "symbol assignment," where some symbol is assigned the value of some expression. The expression may contain other symbols, but it should be restricted to only using symbols that have been assigned values (either as input to the calculator or by an earlier step). Logging is another important kind of step that algorithms might need, they're typically used for providing user-feedback for debugging, general information, warnings, errors, etc. Logging, similar to assignment steps, are provided an expression, for which are evaluated, but the difference is that the logging step displays the value to the user-feedback mechanism of the calculator rather than assigning the value to a symbol. For now, we choose to think of calculators as only having these 2 kinds of steps.

The steps rely on a single mathematical expression language that describes how values can be altered in a predictable way. We choose to restrict our expression language to the common first-order definite-valued mathematical language that most undergraduates in a first-year university mathematics class are familiar with.

Similar to how we expect our expression language to provide predictable results, we also expect our calculators to be predictable and reliably produce accurate results. To gain confidence in accuracy and predictability, we can test our calculators against well-known outputs for well-known inputs. In other words, we can create test cases to gain confidence that our calculator is reliable (which we want to reasonably assure ourselves).

Thus, together, we define our calculator as having 3 main aspects: (a) a set of symbols, (b) an algorithm that somehow calculates the output symbols from the inputs, and (c) a set of test cases to gain confidence that (b) is reliable. To model our calculator DSL, we chose to use Emfatic (EMF) because we found it to be faster to work with. In particular, we found it to be more readable and easier to modify compared to Ecore.

2.2 Emfatic

Listing 1: Calculator EMF

```
class Calculator extends Described {
  attr String[1] name = "";
  val SymbolDeclaration[*] symbolDeclarations;
  val CalculationStep[*] steps;
  val TestCase[*] testCases;
```

```
}
```

When translating the calculators to Emfatic (Listing 1), we also added the name and a description of the calculators using string attributes[3]. The calculator class contains 3 other "containment references" (`vals`): symbol declarations, calculation steps (denoting the algorithm), and a set of test cases.

Symbol declarations (`SymbolDeclaration`, Listing 3) define a symbol (`Symbol`) with a textual description and the *kind* of symbol it is — input (user-provided at the start of the calculator), intermediate (derived from the inputs but not important), or output symbols (symbols that users are interested in calculating). Since the difference between the symbol kinds is quite surface level (or so we claim), we chose to use an `enum` attribute in our symbol declaration type. If there were any behavioural differences, we would have used a subclass. The symbol content itself is contained in another class (`Symbol`) with a one-to-one relationship with symbol declarations. `Symbol`s contain a name attribute (assumed to be a Java-friendly identifier) and a declared *type* of the symbol. The types are containment references of `Type`s (Listing 6).

We chose to build a basic unambiguous, static, judgment-based [5] type system. We restricted our type universe in our calculator to "simple" first-order types with no parameters. Thus, we have booleans, strings, integers, and real numbers. However, we do intend to implement vectors as well. We used an abstract `Type` class to create a generic class of types (with basic feature requirements), and then a subclass (`PrimitiveType`) to designate primitive types using an enumeration of primitive types (`Primitive`). By extending the basic `Type` class with other types, we can further extend our DSLs type system.

The next major component of our calculators are the calculation steps. As already mentioned in Section 2.1, we restricted ourselves to 2 kinds of steps: assignment steps, and print steps. Assignment steps (`AssignmentStep`) and print steps (`PrintStep`) are both subclasses of the abstract base class: `CalculationStep` (Listing 4). Print steps contain a description (we called it a "preface") for the need to print a particular symbol (noted by reference). `AssignmentStep`s contain a reference to a symbol, an optional description `String` attribute, and a value reference to a defining expression. The expression is expected to be well-typed, but we only impose this restriction in Section 4.

Expressions have a shared base class: `Expression` (Listing 7). There are various kinds of expressions we've implemented, such as literals (`TextLiteral`, `BooleanLiteral`, `IntegerLiteral`, `RealLiteral`), symbol references (`SymbolExpression`), unary expressions (`UnaryExpression`), binary expressions (`BinaryExpression`), and tertiary expressions (`TertiaryExpression`). The `Expression` abstract base class defines a basic functionality set that each subclass of expression must provide — namely printing, type inference, symbols it depends on, and a Java representation method.

The last component of our calculators are test cases. We expect that instances of these calculators to obey a set of test cases. Our `TestCase` class is defined with a name and description String attributes, a list of assignment steps (that are expected to reference no symbols in their expressions), and a list of boolean-typed expressions[4].

Finally, these 3 important classes define our calculators, and we may continue to our next project: building a concrete syntax for our DSL.

[3]Note that since we commonly use comments and descriptions, we added a "Described" base class to quickly attach description attributes (Listing 2).

[4]Both restrictions are only imposed in Section 4.

3 Integrated Development Environment

4 Model Validation

Using the Epsilon Validation Language (EVL), we built a series of constraints to impose restrictions we weren't able to adequately impose through the metamodel (Listing 8). In particular, we built a series of constraints (hard requirements) and critiques (soft requirements/preferences) for models of our metamodel.
Calculator Constraints:

1. *named*: each calculator must have a name for users to disambiguate between calculators and their other files,
2. *unique symbols*: each calculator must have a set of symbols that have unique names within that set so variable references can be unambiguous,
3. *at least 1 input*: each calculator should have at least 1 input symbol or else there is no interactivity with the calculators generated, so a normal calculator should be used instead,
4. *at least 1 output*: each calculator should have at least 1 output symbol or else there would be no way to audit or use the calculator,
5. *all outputs assigned once*: all designated output symbols are meaningful in some way and should be guaranteed to have a value at the end of the calculation steps,
6. *all intermediate symbols assigned once*: if a declared intermediate symbol is never assigned, then it can be removed to de-clutter the calculator,
7. *all assignment steps well-typed*: each assignment should involve a symbol of the same type as an expression,
8. *all test case assignments are input symbols*: the pre-calculation state of the calculators should only have input variables assigned values,
9. *all test case assignments contain no variable references*: all assignment bodies in test cases should be evaluable to literal-valued expressions (e.g., contain no symbol references) because there is no defined variables before input set,
10. *all test cases assign all input variables*: each test case should mimic users fully, by providing all necessary inputs the calculator would need,
11. *all test case inputs unambiguous/"no duplicates"*: each defined test case should have no ambiguous symbol assignments or else the initial state of the test case simulations would be ambiguous,
12. *all test case assertions are boolean-typed*: the assertions should be propositional statements or else they should be part of the calculation steps of the main calculator,
13. *all steps depend on non-null symbols*: each step of the calculation should only rely on symbols with defined values or else invalid and undefined expressions and operations will occur, and
14. *all test cases satisfied*: the calculation steps should be simulated under each test case, and each test case should be satisfied.

Calculator Critiques:

1. *description*: it would be nice to have a description of the intent of the calculator and general information about how it works,
2. *all inputs described*: it would be nice for each input symbol to have a user-friendly description to describe what the variable means,
3. *all outputs described*: same as above, and
4. *all test case assertions meaningful*: each test case assertion statement should reference symbols in that calculator or else the assertion would be evaluable without simulation.

Symbol constraints:

1. *named*: each symbol should have a name or else using them would be impossible, and
2. *named with only appropriate characters*: a restriction imposed by desiring Java-compilation and non-unicode data entry — names/identifiers should be usable as Java identifiers.

Finally, we have our last constraint: each expression should be well-typed, or else invalid states are possible, and we don't want calculators to ever get "stuck."

Together, these make up our constraints. We build them using standard EVL tooling (Listing 8) and run them using a Gradle script on our parsed model. If all constraints and critiques are satisfied for a model, then we claim that the produced artifacts (see Section 5) and the model are relatively high-quality. Furthermore, we claim that the produced code artifact should be well-typed, and thus never run into invalid states, errors, or any issue at runtime (excluding that related to business logic).

The simulation of the test cases is one of our larger validation schemes. We had to emulate the Java code that would be run under the test cases provided. Unfortunately, we were not able to add an "evaluation" method to the base `Expression` class because we needed the return type of the function to be `Any`. We wanted to use `Any` because we wanted to use the dynamic typing nature of EOL to avoid model duplication for literal values. However, the EMF language does not support `Any`, and thus we had to "attach" this functionality post-facto in our shared "program.eol" file used by all of our other files. For simulating the calculation steps, we faced similar issues with our base `TestCase` type. We had to have an implicitly known base property of test cases that they are simulatable. For expressions, our evaluation methods were given an (implicitly) read-only "scope" Map for them to reference symbols. Similarly, for our simulation, we also had to pass along the current variable state of the simulation memory. However, the calculation steps are allowed to read from and modify the scope. The full code for simulation works by populating an initial memory state with the input variables, executing the calculation steps on the memory, and then checking that all the provided assertion propositions hold true (Listing 9, specifically the `isSatisfied` operation).

Another notable constraint is the dependency-checking of each calculation step — that is, each step should only rely on symbols that are reasonably guaranteed[5] to have been assigned a value already. In order to validate, we had to iterate over the calculation steps sequentially, processing what impact they would have on symbols and what symbols they rely on to decide whether they are okay or not. The initial set of valid usable symbols are only the input symbols, and it is expected that by the end, each output symbol is calculated.

[5] "Division by 0," and other issues that would require a dependent type system to mitigate are not considered here.

5 Model Management Operations

6 Reflection and Concluding Thoughts

References

[1] Richard Paige. *CAS 703 - Software Design*. A course at McMaster University during the Winter 2023 semester. 2023 (cit. on p. 2).

[2] Microsoft Corporation. *Microsoft Excel*. 2023. URL: https://www.microsoft.com/en-ca/microsoft-365/excel (cit. on p. 2).

[3] Eclipse Foundation, Inc. *Eclipse Epsilon*. 2023. URL: https://github.com/eclipse/epsilon (cit. on p. 2).

[4] Eclipse Foundation, Inc. *Xtext*. 2023. URL: https://github.com/eclipse/xtext (cit. on p. 2).

[5] nLab authors. *judgment*. https://ncatlab.org/nlab/show/judgment. Revision 32. 2023-04 (cit. on p. 4).

Appendix

Listing 2: Abstract Class: Described

```
abstract class Described {
  attr String[1] description = "";
}
```

Listing 3: Symbol Declarations

```
class SymbolDeclaration extends Described {
  attr SymbolDeclarationKind[1] kind;
  val Symbol[1]#declaration symbol;
}

enum SymbolDeclarationKind {
  INPUT;
  INTERMEDIATE;
  OUTPUT;
}

class Symbol {
  id attr String[1] name = "";
  val Type[1] type;
  ref SymbolDeclaration[1]#symbol declaration;
}
```

Listing 4: Calculation Steps

```
abstract class CalculationStep extends Described {
  op String[1] toHRN();
  op Symbol[*] dependencies();
  op Symbol[*] calculates();
  op String[1] toJava();
}

class AssignmentStep extends CalculationStep {
  ref Symbol[1] symbol; // LHS
  val Expression[1] body; // RHS
}

class PrintStep extends CalculationStep {
  attr String[1] preface = "";
  ref Symbol[1] symbol;
}
```

Listing 5: Test Case

```
class TestCase extends Described {
  attr String[1] name = "";
  val AssignmentStep[+] assignments;
  val Expression[+] assertions;
}
```

Listing 6: Type Universe

```
enum Primitive {
  TEXT;
  REAL;
  INTEGER;
  BOOLEAN;
```

```
}

abstract class Type {
  op String[1] serialize();
  op boolean[1] isNumeric();
  op String[1] toJavaType();
}

class PrimitiveType extends Type {
  attr Primitive[1] primitive;
}

class InvalidType extends Type {
  attr String[1] cause;
}
```

Listing 7: Expression Language

```
  abstract class Expression {
  op Type[1] type();
  op String[1] toHRN();
  op Symbol[*] symbolReferences();
  op String[1] toJava();
}

class TextLiteral extends Expression {
  attr String[1] value;
}

class RealLiteral extends Expression {
  attr double[1] value;
}

class IntegerLiteral extends Expression {
  attr int[1] value;
}

class BooleanLiteral extends Expression {
  attr boolean[1] value;
}

class SymbolExpression extends Expression {
    ref Symbol[1] symbol;
}

enum UnaryOp {
  LOGICAL_NOT;
  NEGATION;
}

class UnaryExpression extends Expression {
  attr UnaryOp[1] operator;
  val Expression[1] body;
}

enum BinaryOp {
  // numbers (of same type)
  ADDITION; // also allowed for text and vectors
  MULTIPLICATION;
  DIVISION;
  SUBTRACTION;
```

```
  POWER;
  MODULO;

  // inequalities
  GREATER_THAN;
  GREATER_THAN_EQUAL_TO;
  LESS_THAN;
  LESS_THAN_EQUAL_TO;

  // booleans
  LOGICAL_AND;
  LOGICAL_OR;
  LOGICAL_IMPLIES;

  // any
  EQUIVALENT;
  NOT_EQUIVALENT;
}

class BinaryExpression extends Expression {
  attr BinaryOp[1] operator;
  val Expression[2] exprs;
}

enum TertiaryOp {
  IF_THEN_ELSE;
}

class TertiaryExpression extends Expression {
  attr TertiaryOp[1] operator;
  val Expression[3] exprs;
}
```

Listing 8: Constraints and Critiques

```
operation String withCalculator(c : Calculator): String {
    return c.name + ": " + self;
}

operation allUnique(c : Collection): Boolean {
    var asSet = c.asSet();
    return c.size() == asSet.size();
}

operation Collection hasDuplicates(): Boolean {
    return self.asSet().size() == self.size();
}

context Calculator {
    constraint NameLength {
        check: self.name.length() > 0
        message: "Calculator must have a name."
    }

    critique Description {
        check: self.description.length() > 0
        message: "It would be nice if you defined a human-readable description of your calculator
            ↪ .".withCalculator(self)
    }

    constraint UniqueSymbols {
```

```
    check: allUnique(self.symbols().name)
    message: "All symbols should have unique names.".withCalculator(self)
}

constraint AtLeast1Input {
    check: self.inputs().size() > 0
    message: "Each calculator should have at least 1 input.".withCalculator(self)
}

constraint AtLeast1Output {
    check: self.outputs().size() > 0
    message: "Each calculator should have at least 1 output.".withCalculator(self)
}

critique AllInputsDescribed {
    check: self.inputs().declaration.description.forAll(s|s.length() > 0)
    message: "Each input symbol should have a description for usability.".withCalculator(self
        ↪ )
}

critique AllOutputsDescribed {
    check: self.outputs().declaration.description.forAll(s|s.length() > 0)
    message: "Each output symbol should have a description for usability.".withCalculator(
        ↪ self)
}

constraint AllOutputsAssignedOnce {
    check: self.outputs().forAll(o|self.hasAssignmentStep(o))
    message: "Each output symbol should be assigned at least once in your calculation steps
        ↪ .".withCalculator(self)
}

constraint AllIntermediatesAssignedOnce {
    check: self.intermediates().forAll(o|self.hasAssignmentStep(o))
    message: "Each intermediate symbol should be assigned at least once in your calculation
        ↪ steps.".withCalculator(self)
}

constraint AllAssignmentsWellTyped {
    check: self.assignmentSteps().forAll(astep|astep.symbol.type.equiv(astep.body.type()))
    message: "Each assignment step must be well-typed.".withCalculator(self)
}

constraint AllTestCaseInputSymbolsAreInputs {
    check: self.testCases.forAll(testCase|testCase.assignments.forAll(asgn|asgn.symbol.
        ↪ declaration.kind==SymbolDeclarationKind#INPUT))
    message: "Test case inputs should only assign values to input variables".withCalculator(
        ↪ self)
}

constraint AllTestCaseInputExpressionsLiteral {
    check: self.testCases.forAll(testCase|testCase.assignments.forAll(asgn|asgn.body.
        ↪ symbolReferences().isEmpty()))
    message: "Test case inputs should not reference any symbol in their assignments.".
        ↪ withCalculator(self)
}

constraint AllTestCaseAssignAllInputs {
    // note: using '>=' here because (a) we already ensure that each symbol assigned is an
        ↪ input
```

```
        // and that there are no duplicate assignments.
        check: self.testCases.forAll(testCase|testCase.assignments.symbol.size() >= self.inputs()
            ↪ .size())
        message: "Test case inputs should assign a value to each input.".withCalculator(self)
    }

    constraint AllTestCaseInputsUnambiguous {
        check: self.testCases.forAll(testCase|testCase.assignments.symbol.hasDuplicates())
        message: "Test case inputs may not have ambiguous assignments for input symbols.".
            ↪ withCalculator(self)
    }

    constraint AllTestCaseAssertionsAreBooleanTyped {
        check: self.testCases.forAll(testCase|testCase.assertions.forAll(e|isBoolTy(e.type())))
        message: "Test case assertions must all be propositions (boolean-typed expressions)".
            ↪ withCalculator(self)
    }

    critique AllTestCaseAssertionsMeaningful {
        check: self.testCases.forAll(testCase|testCase.assertions.forAll(e|not e.symbolReferences
            ↪ ().isEmpty()))
        message: "Each test case assertion should have at least 1 symbol reference, or else the
            ↪ check if superfluous".withCalculator(self)
    }

    constraint AllStepsDepsHaveValues {
        check: self.checkStepDependencies()
        message: "Each step's symbol dependencies must be satisfied before that step.".
            ↪ withCalculator(self)
    }

    constraint AllTestCasesSatisfied {
        guard: self.satisfiesAll("AllStepsDepsHaveValues", "AllTestCaseAssertionsAreBooleanTyped
            ↪ ", "WellTyped", "AllIntermediatesAssignedOnce", "AllOutputsAssignedOnce")
        check: self.testCases.forAll(testCase|testCase.isSatisfied(self.steps))
        message: "Not all test cases are satisfied! Check algorithm!".withCalculator(self)
    }
}

context Expression {
    constraint WellTyped {
        check: self.isWellTyped()
        message: "Expression is ill-typed: " + self.type().serialize()
    }
}

context Symbol {
    constraint SymbolsNamed {
        check: Symbol.forAll(s|s.name.length() > 0)
        message: "All symbols must have a non-empty name"
    }

    constraint SymbolsNamedAppropriately {
        check: Symbol.forAll(s|s.name.matches("^[a-zA-Z]([a-zA-Z0-9_]+)?\$"))
        message: "All symbols must start with a letter and be followed by a sequence of letters,
            ↪ numbers or underscores."
    }
}
```

Listing 9: Simulation-specific Code

```
operation TextLiteral evaluate(scope : Map): Any {
   return self.value;
}

operation RealLiteral evaluate(scope : Map): Any {
   return self.value;
}

operation IntegerLiteral evaluate(scope : Map): Any {
   return self.value;
}

operation BooleanLiteral evaluate(scope : Map): Any {
   return self.value;
}

operation SymbolExpression evaluate(scope : Map): Any {
   return scope.get(self.symbol.name);
}

operation UnaryExpression evaluate(scope : Map): Any {
   var value = self.body.evaluate(scope);

   switch (self.operator) {
      case UnaryOp#LOGICAL_NOT:
         value = not value;
      case UnaryOp#NEGATION:
         value = - value;
   }

   return value;
}

operation BinaryExpression evaluate(scope : Map): Any {
   var left = self.exprs[0].evaluate(scope);
   var right = self.exprs[1].evaluate(scope);
   var value = null;

   switch (self.operator) {
     case BinaryOp#ADDITION:
          value = left + right;
     case BinaryOp#MULTIPLICATION:
          value = left * right;
     case BinaryOp#DIVISION:
          value = left / right;
     case BinaryOp#SUBTRACTION:
          value = left - right;
     case BinaryOp#POWER:
          value = left.pow(right);
       case BinaryOp#MODULO:
          value = left.mod(right);
       case BinaryOp#GREATER_THAN:
          value = left > right;
       case BinaryOp#GREATER_THAN_EQUAL_TO:
          value = left >= right;
       case BinaryOp#LESS_THAN:
          value = left < right;
       case BinaryOp#LESS_THAN_EQUAL_TO:
          value = left <= right;
```

```
      case BinaryOp#LOGICAL_AND:
           value = left and right;
      case BinaryOp#LOGICAL_OR:
           value = left or right;
      case BinaryOp#LOGICAL_IMPLIES:
           value = (not left) or right;
      case BinaryOp#EQUIVALENT:
           value = left == right;
        case BinaryOp#NOT_EQUIVALENT:
           value = left != right;
    }

    return value;
}

operation TertiaryExpression evaluate(scope : Map): Any {
    var fst = self.exprs[0].evaluate(scope);
    var value = null;

    switch (self.operator) {
        case TertiaryOp#IF_THEN_ELSE:
            if (fst) {
                value = self.exprs[1].evaluate(scope);
            } else {
                value = self.exprs[2].evaluate(scope);
            }
    }

    return value;
}

operation AssignmentStep simulate(scope : Map): Map {
    scope.put(self.symbol.name, self.body.evaluate(scope));
    return scope;
}

operation PrintStep simulate(scope : Map): Map {
    var stub = self.symbol.name + " = " + scope.get(self.symbol.name).asString();
    if (self.preface?.length() > 0) {
        stub = stub + " ~ " + self.preface;
    }
    stub.println();
    return scope;
}

operation TestCase isSatisfied(progSteps : Collection): Boolean {
    // Create scope map
    var emptyScope = Map{};
    var scope = Map{};
    for (asgn in self.assignments) {
        scope.put(asgn.symbol.name, asgn.body.evaluate(emptyScope)); // note: each input should
            ↪ have an empty scope!
    }
    ("Testing assertion: " + self.name + ", with input set: " + scope.asString()).println();

    // Simulate algorithm steps
    for (step in progSteps) {
        ("Simulating step: " + step.toHRN()).println();
        scope = step.simulate(scope);
    }
```

```
    // Evaluate assertions on the scope and ensure that they are all satisfied
    var allAssertionsSatisfied = true;
    for (assertable in self.assertions) {
        ("Checking assertion: " + assertable.toHRN() + "... ").print();
        if (assertable.evaluate(scope)) {
            "Ok!".println();
        } else {
            "Failed!".println();
            allAssertionsSatisfied = false;
            break;
        }
    }

    if (allAssertionsSatisfied) {
        ("Test case " + self.name + " SUCCEEDED!").println();
    } else {
        ("Test case " + self.name + " FAILED!").println();
    }

    return allAssertionsSatisfied;
}
```