Funcons in Rascal - report

Davy Landman

November 23, 2011

Make sure to chose between sort/non-terminal

Introduction

This document will describe the translation from the ASF+SDF funcons implementation to a Rascal implementation. The first section will summarize the overall architecture, the second section will describe the manual translation to Rascal, the third an automatic generation approach and the last two sections will discuss the limitations and advantages of the Rascal implementation.

1 The architecture

1.1 ASF+SDF

In ASF+SDF the funcons are implemented in a two stage approach. The first stage is translating the CSF (funcon specification files) into SDF definitions. These definitions are used to describe the semantics of a programming language. The funcons (represented as SDF alternatives) are connected to syntax of the programming language using the ASF equations.

Current limitations: At the time of writing, not all funcons are specified in CSF, which means that not all SDF defined funcons were generated from the CSF specifications. Moreover, the CSF specifications contain the descriptions of how to 'interpret' a funcon, but the generation of an interpreter was not yet implemented.

mention MSOS etc?

interpret is incorrect word, should check MSOS papers for better term

1.2 Rascal

The Rascal funcons implementation aims to closely match the ASF+SDF implementation. Rascal however, has more features suited for this domain and we have chosen to use those features to showcase the possible advantages.

The first stage is similar, Rascal also uses CSF to generate the funcon specifications. However, since Rascal has functions and ADTs we do not generate funcon as a language specification but we generate Rascal files containing funcons as a ADT structure. These funcons ADTs can then be used to define the semantics of a programming language in a similar fashion as in ASF+SDF.

Current limitations: Same as in ASF+SDF.

2 Manual implementation

Since only a subset of the funcons were defined in the CSF specifications we first created a manual translation of the funcons in SDF. While this translation lacks modularity, it does provide an insight of how a Rascal funcons implementation works.

2.1 The funcons ADT

The `src/funcons/funcons.rsc` file contains the complete funcons ADT equivelant to the SDF implementation[1]. Below is a snippet from this file.

```
data Expr
    = \data(Data dt)
    | abs(Patt pattern, Comm command)
    | application(Op op, list[Expr] exprs)
    | assignVar(Expr var, Expr val)
    | bound(Id id)
    | deref(Expr var)
    | derefIfVar(Expr var)
    | env(Decl decl)
    | newVar(Expr typeExpr)
    | newVar(Type \type)
    | op(list[Expr] expressions)
    | tup(list[Expr] tuples)
    | tupSeq(list[Expr] tuples)
    | typed(Expr expr, Type check)
    ;
```

2.1.1 `Op` funcon

The `Op` funcon causes the biggest difference in implementation, since the SDF definition creates a structure not easily represented in the ADT structure of RASCAL (this funcon is an example of a funcon only defined in SDF).

Take for example the funcon `int-plus` which can be used used as `int-plus( Expr[[EXP1]], Expr[[EXP1]])`, but also as `comp(int-plus, int-neq)`.

We could solve this by adding the `int-plus` (and all the others) to the `Data` and `Expr` sorts and the `Op` sort, but this would polute our ADT definition. So we chose to only add `int-plus` to the `Op` sort and add a `application` alternative to the `Data` and `Expr` sorts. This would mean we would have to write the first example as: `application(intPlus(), [Expr(EXP1), Expr(EXP2)])`.

However, RASCAL has rewrite rules, and this allows us to write `intPlus(Expr(EXP1), Expr(EXP2))`. It is important to note that we asume to know the amount of parameter usually used for the funcon, else we would have to wrap the parameters into a list.

2.1.2 Explicit production chains

In the ASF+SDF implementation, production chains are often used. This is not visible for the funcons users since ASF+SDF allows implicit chaining when they cause no ambiguity. RASCAL does not support implicit chaining, and this makes a funcon such as `Expr[[ NatCon ]] = NatCon` in ASF becomes `...= data(int(nat(Nat::natCon(val))))` in RASCAL. We can solve this with a rewrite rule, but it means that for each funcon we have to determine where it is actually used and create a rewrite rule for that usage.

2.2 Semantic description using funcons

The `src/lang/pico/semantic/Main.rsc` file contains the semantic description of the pico language, similar to the semantic description of the pico language in ASF+SDF. Below are two definitions, the first is for the loop command, and the second is for the add expression.

```
public Comm pico2Comm(Statement::loop(con, body)) =
    whileLoop(
        noteq(pico2Expr(con), natCon(0)),
        pico2Comm(body)
    );
public Expr pico2Expr(add(lhs, rhs)) = intPlus(pico2Expr(lhs), pico2Expr(rhs));
```

[1] Since the minus character is not allowed in RASCAL names, we used camel casing.

2.2.1 Naming collisions

In the example we see that the Patterns (pattern based invocation) use the qualified name for a alternative, this is due to naming collisions with the funcon names. The AST contains the `loop` alternative for the `Statement` sort, but the `Comm` funcon also defines the `loop` funcon. In most places RASCAL correctly handles these collisions, but at the moment the patterns require fully qualified names to avoid collisions.

2.2.2 List deconstruction

how is this called again?

The only real elegance difference between the RASCAL semantic description and the ASF+SDF description is caused by a current limitation of rascal pattern matching to not allow to match an empty list. The following snippet demonstrates the required code needed to destruct a list:

```
public Comm pico2Comm(list[Statement] statements) =
    (statements == [] ) ? skip() : seq([ pico2Comm(s) | s <- statements]);
```

In a future version of RASCAL we would prefer to have support for the empty list/set/map in the patterns for function parameters[2]. Instead of a head/tail deconstruction of the list we use a comprehension to map the AST to the correct funcons.

2.2.3 AST instead of CST

For our implementation we chose to write the semantic definition against the AST of the pico language, instead of the CST as the ASF+SDF solution does. However, this is not due to a limitation of RASCAL, below is a snippet of how the `pico2Expr(add(lhs, rhs))` would look if written for the CST.

```
public Expr pico2Expr(`<Expression lhs> + <Expression rhs>`)
    = intPlus(pico2Expr(lhs), pico2Expr(rhs));
```

3 CSF generated implementation

Since we have demonstrated that all the funcon concepts from ASF+SDF can be translated to RASCAL, we explain in this section how we could use RASCAL to generate the funcons based on the CSF specifications.

3.1 CSF

The CSF file defines the funcon, it describes it in natural language and it formalizes its semantics using MSOS. Below is the definition of the `decl` funcon. We see that the funcon is part of the `Decl` funcons and takes a `Comm` as parameter.

```
Decl ::= decl(Comm)

Glossary:

This allows $Comm$ to be executed in a sequence of declarations,
computing the empty environment.

Alias:  declaration

1:      Comm --> Comm'
        ---
        decl(Comm) --> decl(Comm')

2:      decl(skip) --> map-empty

3:      Comm ==> Comm'
        ---
        decl(Comm) ==> decl(Comm') : map-empty
```

We rewrote the SDF grammar definition to a RASCAL grammar definition, this grammar is located in the `src/csf2rascal/lang/csf/cst/MainCSFGrammar.rsc` file. The only real difference is that we marked the words such as "Glossary" as keywords, this is important for disambiguation and provides the IDE with information for simple highlighting.

[2] Regular patterns already support this

Since RASCAL provides the `implode` function which automatically maps a CST onto a AST, we defined the AST for CSF and used that AST to write our generator. The AST is defined in the `src/csf2rascal/lang/csf/ast/Main.rsc` file.

3.2 RASCAL funcons generation

Similar to the ASF+SDF approach (`CSF-to-SDF.asf`) we generate RASCAL files from within RASCAL. The `src/csf2rascal/lang/csf/generation/Generate.rsc` file implements this generation in 249 LOC (blank lines excluded).

Unlike the manual implementation, the generated funcons have the same modularity as the ASF+SDF implementation. A user of the funcon can import just the needed funcons. We have added a few small features in comparison to the ASF+SDF generation. We added support for the aliased funcons, and we used RASCAL's `@doc{}` source code annotations to add the informal description of a funcon.

Below is the RASCAL implementation of the above CSF file.

```
extend csf2rascal::generated::Decl::Decl;
import csf2rascal::generated::Comm::Comm;

@doc{This allows Comm to be executed in a sequence of declarations, computing the
 empty environment.}
data Decl = decl(Comm comm);
public Decl declaration(Comm comm) =
    decl(comm);
```

Limitations: We have not solved the problems caused by the explicit production chains since we cannot translate all the funcons automatically. Moreover, since the `Op` funcon was not described in a CSF specification, we have not generated rewrite rules to allow easier usage of these funcons.

3.3 Semantic description using generated funcons

Since we cannot generate all the funcons from CSF files we have no semantic description of the Pico language using these funcons. Moreover, when manually comparing the generated ADT with the manual ADT we see that they are the same. Therefore the semantic description would be equal.

4 Limitations of RASCAL implementation

This section will provide a list of the limitations we found for implementing the funcons in RASCAL.

1. **No implicit chains in the ADTs,** this imposes either a challenge for the CSF to RASCAL generator or the user of the funcons. We suggest that with some smart work in the RASCAL generator this problem can be solved.

2. **Naming collisions,** in the current version some naming collisions could cause confusions for the funcons user. We aim to improve these naming collisions and provide better error messages. These efforts are connected to our project in adding a static type checker.

5 Advantages of RASCAL implementation

This section will provide a list of the advantages we found and envision for implementing the funcons in RASCAL.

1. **Simpler generator,** since one of RASCAL's aims is to support code generation, there are more concepts built into the language than available in ASF+SDF. Moreover, RASCAL includes has native support for regular expressions and has an extensive library for common string operations.

2. **Simple CST to AST transformation,** to avoid repeating the syntactical concepts of CSF, or the language parsed, RASCAL offers a simple way to implode the CST to a AST. This create better maintainable code without the extra maintenance of this transformation.

3. **Eclipse IDE,** RASCAL takes advantage of many features of the Eclipse IDE, this implies that it is easy to provide the users of funcons with a rich user experience.

4. **Writing the interpreter in RASCAL,** it would be possible to write the funcons interpreter using RASCAL. For a example of the possibilities see the `std/library/demo/lang/MissGrant/` folder.