Funcons in Rascal - report

Davy Landman

November 24, 2011

Perhaps include a short table with terms?

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 funcons as language specifications 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

We first created a manual translation of the SDF funcons, since only a subset of the funcons were defined in the CSF specifications. While this translation lacks modularity, it does provide an insight of how a RASCAL funcons implementation could work.

2.1 The funcons ADT

The `src/funcons/funcons.rsc` file contains the complete funcons ADT equivalent to the SDF implementation[1]. Below is a snippet from this file, which shows the CSF `Expr` sort.

```
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 Funcons in `Op`

The funcons in `Op` causes the biggest difference in implementation, since the SDF definition creates a structure not easily represented in the ADT structure of RASCAL (these funcons are an example of funcons 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 pollute 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's rewrite rules allows us to write `intPlus(Expr(EXP1), Expr(EXP2))`. It is important to note that we assume to know the amount of parameter used by 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 funcon users since ASF+SDF allows implicit chaining. RASCAL does not support this, which makes funcon usage as in `Expr[[ NatCon ]] = NatCon` become `... = 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. We expect this has a fairly simple solution.

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 in the parameters of a function use the qualified name for a constructor, this is due to naming collisions with the funcon names. The AST contains the `loop` alternative for the `Statement` non-terminal, but the `Comm` sort 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 these collisions. We expect these issues to be solved when the static type checker is integrated.

2.2.2 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

We have demonstrated that all the funcon concepts from ASF+SDF can be translated to RASCAL in the previous section. We will now explain how we could use RASCAL to generate the funcons from 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 style notation. Below is the definition of the `decl` funcon. We see that the funcon is part of the `Decl` sort 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 in RASCAL, this grammar can be found in the `src/csf2rascal/lang/csf/cst/Mai` 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.

Since RASCAL provides the `implode` function which automatically transforms 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 using 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 funcon user can import only the needed funcons. We have added a two small ‘features’ compared 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.

```
module csf2rascal::generated::Decl::decl_Comm::decl

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` sort was not described in a CSF specification, we have not generated rewrite rules to allow easier usage of those funcons.

3.3 Semantic description using generated funcons

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

4 Limitations of Rascal implementation

This section will provide a list of the limitations we found while 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 can cause confusions for the funcon user. We aim to improve these naming collisions and provide better error messages. These efforts are related to our project for finishing the static type checker.

5 Advantages of Rascal implementation

This section will describe advantages we found and envision when 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 has native support for regular expressions and has an extensive library for common string, set, list, and map 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, such as syntax highlighting and outlining.

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.

5. **Writing a compiler in Rascal,** it would also be possible to write a funcons compiler using Rascal. For a example of the possibilities see the `std/library/demo/lang/MissGrant/` folder or check our Oberon0 compiler[2] which compiles to Java, JVM bytecode, and C.

[2] `http://svn.rascal-mpl.org/oberon0/trunk/`