Funcons in Rascal - report

Davy Landman

November 19, 2011

Introduction

This document will describe the translation from funcons implemented in ASF+SDF to the implementation in Rascal. The first section will summerize the overall architecture, the second section will describe the manual translation to rascal, the third a generative approach and the last two sections will discuss the differences between the two implementations.

1 The architecture

1.1 ASF+SDF

In ASF+SDF the funcons are implemented in a two stage approach. The first stage is 'compiling' the CSF (funcons specifications) into SDF definitions. These definitions can then be used to describe the semantics of a programming language. These funcon semantics 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 the 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?

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 improvements.

The first stage is similiar, Rascal also uses CSF to generate the funcon specifications. However, since Rascal features functions and ADTs we do not generate funcon as a language specification but we generate Rascal files containing the funcon ADT structure. These funcon ADTs can then be used to define the semantics of a programming language in a similar fashion as in ASF+SDF. In our example we chose to define the funcons for the AST of a language, but this is no limitation of Rascal and we could also defined funcons for the CST of a language.

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.

3 CSF generated implementation

4 Limitations of rascal implementation

5 Advantages of rascal implementaton