Rascal Requirements and Design Document

Paul Klint
Tijs van der Storm
Jurgen Vinju

Table of Contents

Note

This document is a braindump of ideas. See the section called "*Issues*"[25] for the issues that have to be resolved.

Introduction

Rascal is the working name for a new version of ASF+SDF, which will be extended with an efficient built-in set and relation data-type. This basically means that we include most features of the RScript language into ASF+SDF. The goals of this language are:

- Separating pure syntax definitions (SDF) from function definitions.
- Easy syntax-directed analysis of programming languages.
- Easy fact extraction.
- Easy connection of fact extraction with fact manipulation and reasoning.
- Easy feedback of analysis results in source code transformation.
- Efficient and scalable implementation.

The above goals are all but one already met in the current design of ASF+SDF, and the current design of RScript. What is missing is the connection (and to be honest: an efficient implementation of relational operators). Alas, any bridge between the two languages is both complex to manage and an efficiency bottleneck. This work is an attempt to consolidate this engineering trade-off.

In the section called "*Integration with Tscripts (Outdated)*"[22] we will also explore the issues when we take integration one step further and also include Tscripts in the considerations.

Requirements

- R1: Runtime speed: large-scale analysis of facts is expensive (frequently high-polynominal and exponential algorithms). A factor speedup can mean the difference between a feasible and an unfeasible case.
- R2: Backward compatible with ASF+SDF. We need to port old ASF+SDF definitions to Rascal.
- R3: Compilation speed: parsetable generation is a major bottleneck in current ASF+SDF. This needs to be fixed.
- R4: Concrete syntax: for readability and easy parsing of a wide range of source languages.
- R5: Functional (no side-effects).
- R6: File I/O (contradicts R5).
- R7: Easily accessible fact storage (similar to a heap, but remember R5 and the details of backtracking).
- R8: List matching (because of R2, influences R7).
- R9: Nesting of data-structures: relations can be nested to model nested features of programming languages (such as scoping).
- R10: Syntax trees can be elements of the builtin data-structures (but not vice versa).

- R11: Features are orthogonal: try to keep the number of ways to write down a program minimal

Mapping features to datatypes

Emphasized cells indicate a new datatype/feature combination that needs to be thought out.

Table 1. Features vs datatypes

Which features work on which datatypes?	CF syntax trees	CF syntax lists	Lexical syntax trees	Lexical syntax lists	Lists	Sets	Relations	Tuples
Pattern matching	Y (CS)	Y, CS, LM	Y, PS	Y, PS, LM	Y, HT	Y, HT	Y, HT	Y
Pattern construction	Y, CS	Y, PS	Y, PS	Y	Y, HT	Y, HT	Y, HT	Y
Generator/ Comprehension	N	N	N	N	LC	SC	SC	N
Complete Functions	Y	Y	Y	Y	Y	Y	Y	Y
Equations	Y, BC	Y, BC	Y, BC	Y, BC	Y	Y	Y	Y
Polymorphism	N	N	N	N	Y	Y	Y	Y
Serialization	AsFix	AsFix	AsFix	AsFix	Y	Y	Y	Y
Traversal Functions	Y	Y	Y	Y	Y	Y	Y	Y
Subtyping	N	N	N, except character class inclusion	N	Y	Y	Y	Y

- BC = Backward Compatible with ASF
- CS = Concrete Syntax
- HT= head/tail matching
- LC = List comprehension
- LM = List Matching
- N = No
- PS = Prefic Syntax
- SC = Set Comprehension
- Y = Yes

Questions

- Q1: What about the overlap between sets of tuples and relations? We should build relations in for efficiency. Is there going to be a type equivalence. Are tuples going to be first class datatypes?
- Q2: Do we need to introduce the basic datatypes Integer and Boolean because sets and relations have typical builtin functions for that? Or can we put these things in a library? How about Locations? We need to introduce as few builtin syntax as possible to prevent ambiguity and confusion... Can

we offer efficient relations without introducing a dependency on a fixed Integer/Boolean syntax??? It would be SIMPLE if only syntax trees are the leafs of nested lists/sets/relations/tuples.

- Q3: Concrete syntax will introduce problems with the new fixed syntax builtin datatypes (prefix functions, lists, sets, relations, tuples). Efficiently make a type-checker that can resolve most of the ambiguities.
- Q4: We should allow (optionally) quoting/escaping concrete syntax parts. If Q3 results in error or undecidable disambiguation, the user must be able to clarify precisely.

Rascal at a glance

Rascal is summarized in the following subsections.

See the section called "*Issues*" [ ] for the issues that still have to be resolved.

Modules

- "Syntax modules" are identical to standard SDF modules and define concrete syntax.
- Ordinary modules may import other modules and define types and functions.

Types

The type system (and notation) are mostly similar to that of Rscript, but

- Symbols (as defined by a syntax module) are also types.
- There are built-in types (bool, int str, loc) that have a syntactic counterpart (not yet defined how to do this exactly).
- Relations can have optional column names.

As a design strategy we try to offer the option to leave out as many type indications as possible.

A type declaration may introduce an abbreviation for a comples type.

Functions

A function declaration consists of a function name, typed arguments, result type and a function body.

A function body contains a number of variable declarations (with optional initializations) followed by an expression. The value of the expression is the value of the function.

Patterns

There is a notation of "pattern": a quoted concrete syntax fragment that may contain variables and subexpresions to be evaluated. We want to cover the whole spectrum from maximally quoted patterns that can unambiguously describe **any** syntax fragment to minimally quoted patterns as we are used to in ASF+SDF. Therefore we support the following mechanisms:

- Optionally typed variables, written as `<TYPE NAME>` or `<NAME>`.
- Quoted patterns enclosed between `[|` and `|]`. Inside a fully quoted string, the characters `<`, `>` and `|` can be escaped as `\<`, `\>`, `\|`. Fully quotred patterns may contain variables.
- Unquoted patterns are an (unquoted) syntax fragment that may contain variables.

Quoted and unquoted patterns form the *patterns* that are supported in Rascal.

Examples are:

- Quoted pattern with typed variables:

```
[| while <EXP Exp> do <${STATEMENT ";"}* Stats> od |]
```

- Quoted pattern with untyped variables:

```
[| while <Exp> do <Stats> od |]
```

- Unquoted pattern with typed variables:

```
while <EXP Exp> do <${STATEMENT ";"}* Stats> od
```

- Unquoted pattern with untyped variables:

```
while <Exp> do <Stats> od
```

Obviously, with less quoting and type information, the probability of ambiguities increases. Our assumption is that a type checker can resolve them.

Implementation hint. For every sort S in the syntax definition add the following rules:

```
S                     -> Pattern
"<" "S"? Variable ">" -> S
```

Expressions

Expressions correspond roughly to Rscript expressions with some extensions:

- There are lists, sets and relations together with comprehensions for these types.
- A "switch" expression is similar to a Switch statement in C or Java and corresponds to the matching provided by the left-hand sides of a set of rewrite rules. However, it provides **only** matching at the top level of its argument and does not traverse its argument.
- A "replace" expression corresponds to a traversal function (transformer). Given a term and a list of pattern/replacement pairs it traverses the term and performs replacements. Note that replace does **not** provide the fixed-point functionality of innermost rewriting (try to rewrite as long as there is a change). This behaviour has to be made explicit by recursion.
- Pattern/replacement pairs may have conditions preceeded by the keyword `when`.
- Generators in comprehensions may range over syntax trees.
- Replace and generators may have a strategy option to indicate:
  - all = continue
  - first = break
  - td = top-down
  - bu = bottom-up

Examples

Here we list experimental examples of Rascal code to try out features.

Tree traversal

Here is the TREE example that we use in explaining traversal functions in ASF+SDF.

```
module Tree-syntax
imports basic/Integers

exports
  sorts TREE
  context-free syntax
    Integer         -> TREE
    f(TREE,TREE)    -> TREE
    g(TREE,TREE)    -> TREE
    h(TREE,TREE)    -> TREE
    i(TREE,TREE)    -> TREE
```

```
module Tree-Examples
imports Tree-syntax

%% Ex1a: Count leaves in a TREE
%% Idea: int N : T generates alle Integer leaves in the tree
%% # is the built-in length-of operator

fun int cnt(TREE T) {
    #{N | int N : T}
}

%% Ex1b: an equivalent,  more purist, version of the same function:
fun int cnt(TREE T) {
    #{N | <Integer N> : T}
}

%% Ex2: Sum all leaves in a TREE
%% NB sum is a built-in that adds all elements in a set or list.
%% Here we see immediately the need to identify
%% - the built-in sort "int"
%% - the syntactic sort "Integer"

fun int sumtree(TREE T) {
    sum({N | int N : T})
}

%% Ex3: Increment all leaves in a TREE
%% Idea: using the construct "replace T { ... }" all leaves in the
%% tree T that match an integer and replaces each N in T by N+1.
%% The expression as a whole returns the modified term.
%% This is an extremely compact manner of writing a transformer!
%% Note that two conversions are needed here:
%% - from int to NAT to match subterms
%% - from int to NAT to convert the result of addition into
%%   a NAT tree

fun TREE inc(TREE T) {
    replace T {
      <Integer N> => <N1>
      when Integer N1 := N + 1
    }
}
```

```
%% Ex4: full replacement of g by i
%% The whole repertoire of traversal functions is available:
%% - replace T first bu { ... }
%% - replace T all td { ... }
%% - etc.
%% with:
%% "first" (= break) and "all" (= continue).
%% "bu" (= bottom-up) and "td" (=top-down)
%% A nice touch is that these properties are not tied to the
%% declaration of a travesal function (as in ASF+SDF) but to
%% its use.

fun TREE frepl(TREE T) {
    replace T all bu{
      g(<TREE T1>, <TREE T2>) => i(<TREE T1>, <TREE T2>)
    }
}

fun TREE drepl(TREE T) {
    replace T first {
      g(<TREE T1>, <TREE T2>) => i(<TREE T1>, <TREE T2>)
    }
}

fun TREE drepl(TREE T) {
    replace T first bu {
      g(<TREE T1>, <TREE T2>) => i(<TREE T1>, <TREE T2>)
    }
}

%% Ex6: shallow replacement of g by i (i.e. only outermost
%% g's are replaced);

fun TREE srepl(TREE T) {
    replace T first td {
      g(<TREE T1>, <TREE T2>) => i(<TREE T1>, <TREE T2>)
    }
}

%% Continuing this line of thought, we can also add these
%% directives to all generators (where "all td" would be
%% the default):

fun set[TREE] find_outer_gs(TREE T) {
    { S | first td STATEMENT S : T,
          g(<TREE T1>, <TREE T2>) := S }
}
```

Booleans

Probably a non-typical example, but let's try it anyway. It looks horrible compared to the ASF version but gives an indication how we can convert ASF+SDF specifications to Rascal. See the section called "*Booleans (version2 using replace)*" [8] for a much shorter version that uses `replace`.

```
module Booleans-syntax
exports
  sorts Bool
```

```
  context-free syntax
    "true"        -> Bool
    "false"       -> Bool
    Bool "&" Bool -> Bool {left}
    Bool "|" Bool -> Bool {right}
```

```
module Bool-examples
imports Booleans-syntax

fun Bool and(Bool B1, B2) { %% reduce & operator
    switch B1 {
      true      => reduce(B2)
      false     => false
      <Bool B>  => <B1> & <B2>
    }
}

fun Bool or(Bool B1, B2) { %% reduce | operator
    switch B1 {
      true      => true
      false     => reduce(B2)
      <Bool B>  => <B1> | <B2>
    }
}

fun Bool reduce(Bool B){
    switch B {
      <Bool B1> & <Bool B2> => and(B1, B2)
      <Bool B1> | <Bool B2> => or(B1, B2)
      <Bool B3>             => B
    }
}
```

Booleans (version2 using replace)

The earlier definition of Booleans was quite involved. A much simpler solution exists using the replace construct that we have encountered in the above examples.

```
module Bool-examples2
imports Booleans-syntax

fun Bool reduce(Bool B) {
    replace B bu {
      true & <Bool B2>   => reduce(B2)   %% Style 1: Variables and recursion
      false & <Bool B2>  => false

      true | true   => true              %% Style 2: truth table
      true | false  => true
      false | true  => true
      false | false => false
    }
}
```

Observe that there are two styles:

- Using variables on the left-hand side: a recursive application of reduce is needed to fully normalize the result.

- A truth table: this is sufficient as is.

Substitution in Lambda

Below a definition of substitution in lambda expressions. It would be nice to get this as simple as possible since it is a model for many binding mechanisms. It is also a challenge to write a generic substitution function that only depends on the syntax of variables and argument binding.

```
module examples/Lambda/Lambda-syntax

exports
sorts Var %% variables
      Exp %% expressions
lexical syntax
     [a-z]+               -> Var
context-free syntax
     "prime" "(" Var ")"  -> Var  %% generate unique name
     Var                  -> Exp  %% single variable
     "fn" Var "=>" Exp    -> Exp  %% function abstraction
     Exp Exp              -> Exp  %% function application
```

Examples:

```
module Lambda-Examples
imports Lambda-syntax

fun set[Var] allVars(Exp E) {
    {V | Var V : E}
}

fun set[Var] boundVars(Exp E) {
    {V | fn <Var V> => <Exp E1> : E}
}

fun set[Var] freeVars(Exp E) {
    allVars(E) \ boundVars(E)
}

%% Generate a fresh variable if V does not occur in
%% given set of variables.

fun Var fresh(Var V, set[Var] S) {
    if V in S then prime(V) else V fi
}

%% Substitution: replace all occurrences of V in E2 by E1

fun Exp subst(Var V1, Exp E1, Exp E2) {

    switch E2 {
      <Var V2> => V2
         when V1 != V2

      <Var V2> => E1
         when V1 == V2

      <Exp Ea> <Exp Eb>  => <Exp EaS> <Exp EbS>
         when Exp EaS := subst(V, E, Ea),
```

```
              Exp EbS := subst(V, E, Eb)

      fn <Var V2> => <Var Ea> =>  fn <Var V2> => <Exp Ea>
         when V1 == V2

      fn <Var V2> => <Exp Ea> => fn <Var V2> => <Exp E1S>
         when V1 != V2,
              not(V1 in freeVars(E2) & V2 in freeVars(E1)),
              Exp E1S := subst(V1, E1, Ea)

       fn <Var V2> => <Exp Ea>  => fn <Var V3> => <Exp EaS>
         when V1 != V2,
              V1 in freeVars(Ea) & V2 in freeVars(E1),
              V3 := fresh(V2, freeVars(Ea) union freeVars(E1)),
              Exp EaS := subst(V1, E1, subst(V2, V3, E2))
    }
}
```

Note

There are a lot of ambiguities in the body of subst!

Renaming in Let

```
module Let-syntax
exports
sorts Var %% variables
      Exp %% expressions
lexical syntax
     [a-z]+                            -> Var
context-free syntax
     Var                               -> Exp
     "let" Var "=" Exp "in" Exp "end" -> Exp
```

Examples:

```
module Let-Example
imports Let

%% Rename all bound variables in an Exp
%% Version 1: purely functional
%% Exp: given expression to be renamed
%% rel[Var,Var]: renaming table
%% Int: counter to generate global variables

fun Exp rename(Exp E, rel[Var,Var] Rn, Int Cnt) {
    switch(E) {
    let <Var V> = <Exp E1> in <Exp E2> end =>
           let <Var Y >= <Exp E1R>
           in
              <Exp E2R>
           end
        when Var Y := "x" + Cnt,  %% this + operator concatenates
                                  %% (after converting the int to str)
             int Cnt1 := Cnt + 1,
     Exp E1R := rename(E1, Rn, Cnt),
             Exp E2R := rename(E2, {<V, Y>} union Rn, Cnt1)
```

```
    <Var V> => V1
       when { Var V1 } := Rn[V]

    <Exp E1> => E1
    }
}
```

Here are some OUTDATED experiments with global variables:

```
%% Rename all bound variables in an Exp
%% Version 2: using a global variable
%% to generate new variables

fun Var newVar() {
    global int Cnt := 0   %% Initialize global Cnt on first call
                          %% of newVar. This is similar to a
                          %% local static var in C.
    Cnt := Cnt + 1;
    "x" + Cnt
}

%% Rename -- Version 2

fun Exp rename(Exp E, rel[Var,Var] Rn) {
    Var V, Y;

    switch E {

    [| let $V = $E1 in $E2 end |] =>
           [| let $Y = $(rename(E1, Rn))
              in
                $(rename(E2, {<V, Y>} union Rn))
              end
           |]
       when Y := newVar()

    [| $V |] => V1
       when { V1 } == Rn[V]

    [| $E |] => E
    }
}

%% Rename -- Version 3, with Rn also as global variabele

fun Var newVar() {
    global int Cnt := 0
    Cnt := Cnt + 1;
    "x" + Cnt
}

fun Exp rename(Exp E) {
    global rel[Var, Var] Rn := {}
    Var V, Y;

    switch E {
    [| let $V = $E1 in $E2 end |] =>
           [| let $Y = $rename(E1)
```

```
              in
                 $rename(E2)
              end
           |]
       when Y := newVar,
            Rn :=  {<V, Y>} union Rn

    [| $V |] => V1
       when { V1 } == Rn[V]

    [| $E |] = E
    }
}

%% Question: how to reset the value of global variables?
%% Idea: model them as arguments:
%% - fun Var newVar(global int Cnt := 0) { ... }
%% and allow calls without arguments (as in above example) or
%% with arguments:
%% - newVar(13)
%% which resets the value of Cnt.
```

Concise Pico Typechecker

The following example shows the tight integration ASF with comprehensions.

```
module Typecheck

imports Pico-syntax
imports Errors

type Env = rel[PICO-ID,TYPE];

fun list[Error] tcp(PROGRAM P) {
    switch P {
    begin <DECLS Decls> <{STATEMENT ";"}* Series> end =>
           [ tcst(S, Env) | Stat S : Series ]     %% list comprehension
       when Env Env := {<Id, Type> |
                        <PICO-ID Id> : <TYPE Type> : Decls}
    }
}

fun list[Error] tcst(Stat Stat, Env Env) {
    switch Stat {
    <PICO-ID Id> := <EXP Exp> => type-of(Exp, Type, Env)
       when {<Id, Type>} := Env[Id]

    if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                 else <{STATEMENT ";"}* Stats1> fi =>
           type-of(Exp, natural, Env) +
           tcs(Stats1, Env) + tcs(Stats2, Env)

    while <EXP Exp> do <{STATEMENT ";"}* Stats1> od =>
           type-of(Exp, natural, Env) + tcs(Stats, Env)
    }
}

fun list[Error] type-of(Exp E, TYPE Type, Env Env) {
```

```
    switch E {
    <NatCon N> => []
       when Type == natural

    <StrCon S> => []
       when Type == string

    <PICO-ID Id> => []
       when {<Id,Type2>} := Env[Id],
            Type2 == Type

    <EXP E1> + <EXP E2>  => type-of(E1, natural, Env) +
                            type-of(E1, natural, Env)
       when Type == natural

    <EXP E1> + <EXP E2>  => type-of(E1, natural, Env) +
                            type-of(E1, natural, Env)
       when Type == natural

    <EXP E1> || <EXP E2> => type-of(E1, string, Env) +
                            type-of(E1, string, Env)
       when Type == string

    <EXP Exp> => [error("Incorrect type")]
    }
}
```

```
%% Discussion: it may be interesting to generalize switch to allow
%% multiple arguments:
   switch (E, Type) {
    [| $NatCon |], [| natural |] => []

    [| $StrCon |], [| string |] => []

    [| $Id |], Type => []
       when {<Id,Type2>} := Env[Id],
            Type2 == Type

    [| $Exp |], Type => [error("Incorrect type")]
    }
%% Even patterns and expressions could be allowed as cases.
```

Pico control flow extraction

```
module Pico-controlflow
imports pico/syntax/Pico

type CP = EXP | STATEMENT;     %% A Code Point, union of two types

type CFSEGMENT = < set[CP] entry, rel[CP,CP] graph, set[CP] exit>;

fun CFSEGMENT cflow({STATEMENT ";"}* Stats){
    switch Stats {
      <STATEMENT Stat> ; <{STATEMENT ";"}* Stats2> =>
             <En1, R1 union R2 union (Ex1 x En2), Ex2>
         when <set[CP] En1, rel[CP,CP] R1, set[CP] Ex1> := cflow(Stat),
              <set[CP] En2, rel[CP,CP] R2, set[CP] Ex2> := cflow(Stats2)
```

```
      [| |] => <{}, {}, {}>
    }
}

fun CFSEGMENT cflow(STATEMENT Stat){
    switch Stat {
      while <EXP Exp> do <{STATEMENT ";"}* Stats> od =>
             <{Exp}, ({Exp} x En) union R union (Ex x {Exp}),{Exp}>
         when <set[CP] En,  rel[CP,CP] R,  set[CP] Ex> := cflow(Stats)

      if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats2> fi =>
             < {Exp},
               ({Exp} x En1) union ({Exp} x En2) union R1 union R2,
                Ex1 union Ex2
             >
         when <set[CP] En1, rel[CP,CP] R1, set[CP] Ex1> := cflow(Stats1),
              <set[CP] En2, rel[CP,CP] R2, set[CP] Ex2> := cflow(Stats2)

      <STATEMENT Stat>  => <{Stat}, {}, {Stat}>
    }
}
```

Note that most type information in declarations can be omitted, yielding a more friendly version, e.g.

```
fun CFSEGMENT cflow(STATEMENT Stat){
    switch Stat {
      while <EXP Exp> do <{STATEMENT ";"}* Stats> od =>
             <{Exp}, ({Exp} x En) union R union (Ex x {Exp}),{Exp}>
         when <En, R, Ex> := cflow(Stats)

      if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats2> fi =>
             < {Exp},
               ({Exp} x En1) union ({Exp} x En2) union R1 union R2,
                Ex1 union Ex2
             >
         when <En1, R1, Ex1> := cflow(Stats1),
              <En2, R2, Ex2> := cflow(Stats2)

      <STATEMENT Stat>  => <{Stat}, {}, {Stat}>
    }
}
```

Pico use def extraction

```
module Pico-use-def

imports pico/syntax/Pico

fun rel[EXP,PICO-ID] uses(PROGRAM P) {
  {<E,Id> | EXP E : P, PICO-ID Id := E}
}

fun rel[STATEMENT, PICO-ID] defs(PROGRAM P) {
  {<S, Id> | STATEMENT S : P,
                        <PICO-ID Id> := <EXP Exp> := S}
}
```

The above uses a "matching condition" to decompose S. The problem solved is that we want to have a name for the whole assignment *and* for the lhs identifier.

Pico uninitialized variables

```
module Pico-uninit
imports pico/syntax/Pico
        Pico-controlflow
        Pico-use-def

fun set[PICO-ID] uninit(PROGRAM P) {
    rel[EXP,PICO-ID] Uses := uses(P);
    rel[STATEMENT, PICO-ID] Defs := defs(P);
    CFSEGMENT CFLOW := cflow(P);
    set[CP] Root := CFLOW.entry;
    rel[CP,CP] Pred := CFLOW.graph;

    {Id | <EXP E, PICO-ID Id> : Uses,
                E in reachX(Root, Defs[-,Id], Pred)
    }
}
```

Questions (UPDATE THIS):

- There is a subtyping issue here. De type of `reachX` is:

  ```
  set[&T] reachX(set[&T] Start,

  set[&T] Excl,

  rel[&T,&T] Rel)
  ```

  but `E` has type `EXP`, `{ROOT}` has type `set[STATEMENT]`, and `cflow` has type `rel[CP,CP]`, with `type CP = EXP | STATEMENT`. It requires proper subtyping, e.g. set[STATEMENT] < set[CP], to type this.

Pico common subsexpression elimination

```
module Pico-common-subexpression

imports pico/syntax/Pico
        Pico-controlflow
        Pico-use-def

fun PROGRAM cse(PROGRAM P) {
    rel[STATEMENT, PICO-ID] Defs := defs(P);
    rel[CP,CP] Pred := cflow(P).graph;
    rel[EXP, PICO-ID] replacements :=
       {<E2, Id> | STATEMENT S : P,
                   <PICO-ID Id> := <EXP E> := S,
                   Id notin E,
                   EXP E2 : reachX({S}, Defs[-,Id], Pred)
       };

    replace P {
      <EXP E > => Id
         when { Id } := replacements[E]
    }
}
```

Paraphrased: Replace in P all expressions E2 by Id, such that

- P contains a statement S of the form Id := E,
- Id does not occur in E,
- E2 can be reached from S,
- There is no redefinition of Id between S and E2.

Note that a slight abbreviation is possible if we introduce labelled patterns (here S):

```
 rel[EXP, PICO-ID] replacements :=
       {<E2, Id> | <PICO-ID Id> := <EXP E> S : P,
                   Id notin E,
                   EXP E2 : reachX({S}, Defs[-,Id], Pred)
       };
```

Also note that we could factor out the assignment pattern to make cse more generic if we introduce patterns a sfirst class citizens:

```
fun PROGRAM cse(PROGRAM P, pat STATEMENT Assign(PICO-ID Id, EXP E)) {
 ...
    rel[EXP, PICO-ID] replacements :=
       {<E2, Id> | Assign S : P,
                   Id notin E,
                   EXP E2 : reachX({S}, Defs[-,Id], Pred)
       };

 ...
}
```

Example invocations (Pico style)

```
cse(P, <PICO-ID Id> := <EXP E>)
```

or (Cobol style):

```
cse(P, move <EXP E> to <PICO-ID Id>)
```

Note that the order of variables in the pattern and its declaration may differ.

It is to be determined how the instantiation of a pattern looks, e.g.

```
Assign([|x|], [| y = 1 |])
```

Pico constant propagation

```
module Pico-constant-propagation

imports pico/syntax/Pico
        Pico-controlflow
        Pico-use-def

fun Boolean is-constant(EXP E) {
   switch E {
     <NatCon N> => true

     <StrCon S> => true

     <EXP E> => false
```

```
   }
}

fun PROGRAM cp(PROGRAM P) {
    rel[STATEMENT, PICO-ID] Defs := defs(P);
    rel[CP,CP] Pred := cflow(P).graph;

    rel[PICO-ID, EXP] replacements :=
      {<Id2, E> | STATEMENT S : P,
                  <PICO-ID Id> := <EXP E> := S,
                  is-constant(E),
                  PICO-ID Id2 : reachX({S},Defs[-,Id],Pred),
                  Id2 == Id
      };

    replace P {
     <PICO-ID Id> => E
        when { EXP E } := replacements[Id]
    }
}
```

Paraphrased: Replace in P all expressions Id2 by the constant E, such that

- P contains a statement S of the form Id := E,
- E is constant,
- Id2 can be reached from S,
- Id2 is equal to Id,
- There is no redefinition of Id between S and Id2.

Innerproduct

[Example taken from TXL documentation]

Define nnerproduct on vectors of integers, e.g. (1 2 3).(3 2 1) => 10.

```
module examples/Vectors/Vector-syntax

exports
  imports basic/Integers
sorts Vector

context-free syntax
   "(" Integer* ")"    -> Vector
   Vector "." Vector   -> Integer
```

```
module Innerproduct

imports Vector-syntax

fun int innerProduct(Vector V1, V2){
    if ( <Integer N1> <Integer* Rest1> ) := V1 &
       ( <Integer N2> <Integer* Rest2> ) := V2
    then
       (N1*N2) + innerProduct( (<Rest1>), (<Rest2>) )
    else
       0
```

```
    fi
}
```

Bubble sort

[Example taken from TXL documentation]

```
module Bubble

fun Integer* sort(Integer* Numbers){
    replace Numbers {
      <Integer* Rest1> <Integer N1> <Integer N2> <Integer* Rest2> =>
         sort(<Integer* Rest1> <Integer N2> <Integer N1> <Integer* Rest>)
      when N1 > N2
    }
}
```

This example raises a number of issues about the execution of replace.

First, are Rest1 and Rest needed? Would

```
    replace Numbers {
      <Integer N1> <Integer N2> => <Integer N2> <Integer N1>
      when N1 > N2
      <
    }
```

be ok?

Another way to write this is:

```
module Bubble

fun Integer* sort(Integer* Numbers){
    sort( replace Numbers {
            <Integer N1> <Integer N2> => <Integer N2> <Integer N1>
            when N1 > N2
          }
        )
}
```

The replace will replace all adjacent pairs that are in the wrong order in the current list. The resulting list becomes the argument of a new call to sort.

Generic Bubble sort

Here is a generic bubble sort wich uses type parameters (&ELEM) and a function parameter.

```
module Bubble

fun &Elem* sort(Elem* Elements, fun bool GreaterThan(&Elem, &Elem)){
    sort( replace Elements {
            <E1> <E2> => <E2> <E1>
            when GreaterThan(E1, E2)
          }
        )
}
```

Do we want this generality? What are the implications for the implementation? The current syntax does not yet allow type variables in patterns.

Outdated examples

Pico Typecheck using dynamically scoped variables (OUTDATED)

The following example details the use of a dynscope global variable for building up and using an environment. Notice that we introduce *void* functions as a result.

```
module Typecheck

imports Pico-syntax

hiddens

type Env = rel[PICO-ID,TYPE]

exports

fun Bool tc(PROGRAM P) {
    dyn Env Env := {};
    switch P {
      [| begin $Decls $Stats end |] => tcs(Stats)
         when Env := tcd(Decls)
    }
}

fun Env tcd(Decls Decls) {
    {< Id, Type> | [| $(PICO-ID Id) : $(TYPE Type) |] : Decls}
}

fun Bool tcs(Stats Stats) {
    Stat Stat;
    Stats Stats;
    switch (Stats) {
    [| |]               => true

    [| $Stat ; $Stats |] => tcst(Stat) + tcst(Stats)
    }
}

fun Bool tcst(Stat Stat) {
    Id Id;
    Exp Exp;
    Stats Stats, Stats1, Stats2;
    switch (Stat) {
      [| $Id := $Exp |] => type-of(Exp, Type)
         when {<Id, Type>} := Env[Id]

      [| if $Exp then $Stats1 else $Stats2y fi |] =>
         type-of(Exp, natural) + tcs(Stats1) + tcs(Stats2)

      [| while $Exp do $Stats od |] =>
         type-of(Exp, natural) + tcs(Stats)
    }
}
```

```
fun list[Error] type-of(Exp E, TYPE Type) {
    NatCon NatCon;
    StrCon StrCon;
    Id Id;
    Exp Exp;

    switch (E) {
      [| $NatCon |] => []
         when Type == natural

      [| $StrCon |] => []
         when Type == string

      [| $Id |] => []
         when {<Id,Type2>} := Env[Id],
              Type2 == Type

      [| $Exp |] => [error("Incorrect type")]
    }
}
```

Pico eval with dynamically scoped variables (OUTDATED)

A Pico evaluator using dynamic variables. It still uses functions that return VEnvs (this is not consistent and should be changed).

```
fun VEnv evalProgram(Program p) {
    dyn VEnv venv;
    Decls decls;
    Series series;

    switch p {
      [| begin $decls $series end |] => evalStatements(series)
         when venv := evalDecls(decls);
    }
}

fun VEnv evalDecls(Decls decls) {
    Id-Type* idtypes;
    switch decls {
      [| declare $idtypes |] => evalIdTypes(idtypes)
    }
}

fun VEnv evalIdTypes(Id-Type* idtypes) {
    Id id;
    Id-Type* tail;
    switch idtypes {
      [| $id : natural, $tail |] => store(evalIdTypes(tail),id,0)

      [| $id : string, $ tail |] => store(evalIdTypes(tail),id,"")

      [| |] => []
    }
}
```

```
fun VEnv evalStatements(Statement* series) {
    Statement stat; Statement* stats;
    switch series  {
      [| $stat; $stats |] => venv
         when venv := evalStatement(stat),
              venv := evalStatements(stats)
      [| |] => venv
    }
}

fun VEnv evalStatement(Statement stat) {
    Exp exp;
    Series series, series1, series2;

    switch stat {
      [| if $exp then $series1 else $series2 fi |] =>
         evalStatements(series1)
         when evalExp(exp, venv) != 0

      [| if $exp then $series1 else $series2 fi |] =>
         evalStatements(series2)
         when evalExp(exp, venv) == 0

      [| while $exp do $series od |] => venv
         when evalExp(exp, venv) == 0


      [| while $exp do $series od |] =>
         evalStatement([|while $exp do $series od |])
         when evalExp(exp, venv) != 0,
              venv := evalStatements(series)
    }
}

fun VEnv evalExp(Exp exp) {
    Exp exp1, exp2;
    Natural nat1, nat2;
    StrCon str1, str2, str3;

    switch exp {
      [| $exp1 + $exp2 |] => nat1 + nat2
         when nat1 := eve(exp1),
              nat2 := eve(exp2)

      [| $exp1 - $exp2 |] => nat1 -/ nat2
         when nat1 := eve(exp1),
              nat2 := eve(exp2)


      [| $exp1 \|\| $exp2 |] => str3
         when str1 := eve(exp1),
              str2 := eve(exp2),
              str3 := concat(str1, str2)

      [| $exp1 |] => nil-value     %% default "equation"
   }
}
```

Generating Graph files in Dot format

This example illustrates the use of a comprehension on the right-hand side of an equation.

```
module Dot-generation

imports Dot-syntax

fun Dot gen-dot(rel[ID, ID] Rel) {
    [| digraph example {
                        $([ node(Tup) | <ID,ID> Tup : Rel ])
                       }
    |]
}

fun DotElem* gen-node(<ID Id1, ID Id2>) {
    [| node $Id1; node $Id2; $Id1 -> $Id2 |}}
}
```

Syntax Definition

See separate SDF definition.

Integration with Tscripts (Outdated)

Note

It is not yet clear whether we will also include Tscript in the integration effort. For the time being, this section is considered outdated.

Introduction

It is possible to speculate on an even further integration of formalisms and combining the above amalgan of ASF+SDF and Rscript with Tscripts.

Requirements

- R12: The resulting language uses a single type system. This means that relational types (possible including syntactic objects) can be used in Tscripts.
- R13: The current "expressions" in Tscript (terms that occur at the rhs of an assignment) are replaced by calls to ASF+SDF or Rscript functions.
- R14: There is minimal duplication in functionality between ASF+SDF/Rscript/Tscript.

Different styles of Type Declarations

We have at the moment, unfortunately, a proliferation of declaration styles for types.

Functions are declared in ASF+SDF as:

```
typecheck(PROGRAM) -> Boolean
```

Observe that only the type of the parameter is given but that it does not have a name.

In Rscript we have:

```
int sum(set[int] SI) = ...
```

while in Tscript processes are declared as

```
process mkWave(N : int) is ...
```

For variables a similar story applies. Variables are declared in ASF+SDF as:

```
"X" [0-9]+    -> INT
```

In Rscript we have:

```
int X

int X : S   (in comprehensions)
```

and in Tscript we have:

```
X : int
```

In order to unify these styles, we might do the following:

- The type of an entity is always written before the entity itself.
- Formal parameters have a name.

In essence, this amounts to using the declaration style as used in Rscript. So we get:

```
Boolean typecheck(PROGRAM P) is ...

process mkWave(int N) is ...
```

Or do we want things like:

```
function typecheck(P : PROGRAM) -> Boolean is ...

var X -> int

process mkWave(int N) is ...
```

Advantages are:

- The category of the entity is immediately clear (function, var, process, tool, ...).
- It is readable to further qualify the category, i.e., traversal function, hidden var, restartable tool)

Global Flow of Control

We have to settle the possible flow of control between the three entities ASF+SDF, Rscript and Tscript. Since Tscript imposes the notion of an atomic action it would be problematic to have completely unrestricted flow of control. Therefore it is logical to use Tscript for the top-level control and to limit the use of ASF+SDF and Rscript to computations within atomic actions. There is no reason to restrict the flow of control between ASF+SDF and Rscript.

What are the consequences of the above choice? Let's analyze two cases:

- Parse a file from within an ASF+SDF specification. This (and similar built-ins) that use the operating system are removed from ASF+SDF. Their effect has to be achieved at the Tscript-level which is the natural place for such primitives.
- Describe I/O for a defined language. Consider Pico extended with a read statement. Here the situation is more complicated. We cannot argue that the flow of control in the Pico program (as

determined by an interpreter written in ASF+SDF) should be moved to the Tscript level since Tscript simply does not have the primitives to express this. On the other hand, we have to interrupt the flow of control of the Pico interpreter when we need to execute a read statement. The obvious way to achieve this is

- At the Tscript level, a loop repeatedly calls the Pico interpreter until it is done.
- After each call the Pico interpreter returns with either:
  - An indication that the eecution is complete (and possibly a final state and/or final value).
  - An indication that an external action has to be executed, for instance the read statement. This indication should also contain the intermediate state of the interpreter. When the external action has been executed, the Pico interpreter can be restarted with as arguments the value of the external action and the intermediate state.

Experimentation will have to show whether such a framework is acceptable.

Modularization

Rscript and Tscript have no, respectively, very limited mechanisms for modularization. ASF+SDF, however, provides a module mechanism with imports, hiding, parameters and renaming. This mechanism was originally included in ASF, was taken over by SDF and is now reused in ASF+SDF. Currently, there are not yet sufficiently large Rscripts to feel the need for modules. In Tscript, there is a strong need for restricted name spaces and for imposing limitations on name visibility in order to limit the possible interactions of a process with its surroundings and to make it possible to create nested process definitions. What are the design options we have to explore?

First, we can design a new module system that is more suited for our current requirements. The advantage is that we can create an optimal solution, the disadvantage is that there are high costs involved regarding implementation effort and migrations of existing ASF+SDF specifications to the new module scheme.

Second, we can design an add-on to the ASF+SDF module system that addresses our current needs.

Third, we can try to reuse the current ASF+SDF module system.

As a general note, parameterized modules, polymorphics types and renamings are competing features. We should understand what we want. It is likely that we do not need all of them.

Before delving into one of the above alternative approaches, let's list our requirements first.

- We need grammar modules that allow the following operations: import, renaming, deletion (currently not supported but important to have a fixed base grammar on many variations on it). Parameterization and export/hidden: unclear.
- We need function modules (ASF+SDF and Rscript) that provide: import, maybe parametrization, and export/hidden.
- We need process modules (Tscript) that provide: import, export, hiding.

Protyping/implementation of Rascal

Every prototype will have to address the following issues:

- Parsing/typechecking/evaluating Rascal.
- How to implement the relational operations.
- How to implement matching.

- How to implement replacement.
- How to implement traversals.

The following options should be considered:

- Implementation of a typechecker in ASF+SDF:
  - Gives good insight in the type system and is comparable in complexity to the Rscript typechecker.
  - Work: 2 weeks
- Implementation of an evaluator in ASF+SDF.
  - Requires reimplementation of matching & rewriting in ASF+SDF.
  - Bound to be very slow.
  - Effort: 4 weeks
- Implementation of a typechecker in Rascal.
  - Interesting exercise to asses Rascal.
  - Not so easy to do without working Rascal implementation.
  - Not so easy when Rascal is still in flux.
  - Effort: 1 week
- Implementation of an evaluator in Rascal.
  - Ditto.
- Extending the current ASF+SDF interpreter.
  - This is a viable options. It requires extensions of AsFix.
  - Effort: 4 weeks
- Translation of Rascal to ASF+SDF in ASF+SDF.
  - Unclear whether this has longer term merit.
  - Allows easy experimentation and reuse of current ASF+SDF implementation.
  - Effort: 4 weeks
- Implementation of an interpreter in Java.
  - A future proof and efficient solution.
  - Requires reimplementation of matching & rewriting in Java.
  - Effort: 8 weeks.

Issues

- See the section called “*Patterns*”[4] for a description of patterns. There are still some questions about patterns:
  - Do we want the subexpressions in patterns? [Proposal: no since it complicates the syntax]

    - Do we want string variables in patterns? [Undecided]
    - Do we want to add regular expression matching primitives to patterns? Ex.
        - `[| if @any@ $Stats fi |]`
- Dynamic variables need more thought; do we really want them? [Proposal: let's do without them.]
- The relation (no pun intended) between local variable declarations in functions, patterns and comprehensions has to be established.
- We keep the polymorphic types &T1 as in Rscript.
- How does list matching fit in? Probably no problem, see above examples.
- How do we identify built-in sorts (bool, int, etc) with their syntactic counterparts?
- What happens if no case in a switch matches? Some kind of failure? How does it propagate? Runtime error?
- In a list comprehension: do list values splice into the list result?
- Ditto for set comprehensions.
- Unexplored idea: add (possibly lazy) generators for all types; this allows to generate, for instance, all statements in a program.
- The "equations" from Rscript are not yet included.
- Shopping list of ideas in Tom:
    - Named patterns to avoid building a term, i.e. w@[| while $Exp do $stat od |].
    - Anti-patterns, i.e. the complement of a patterns: ! [| while $Exp do $stat od |] matches anything but a while.
    - Anonymous variables a la Prolog: [| while $_ do $stat od |].
    - String matching in patterns.
    - Tom uses the notation %[...]% for quoted strings with embedded @...@ constructs that are evaluated. It also has a backquote construct.
- Shopping list of ideas from TXL:
    - "redefine" allows modification of an imported grammar.
    - An "any" sort that matches anything.