
Rascal Requirements and Design Document

Paul Klint
Tijs van der Storm
Jurgen Vinju

Table of Contents

Introduction	2
Requirements	2
Mapping features to datatypes	2
Questions	3
Rascal at a glance	4
Examples	4
Booleans	4
Deep tree matching instead of tree traversal	5
Booleans (version2 using replace)	8
Dot Product	8
Bubble sort	9
Generic Bubble sort	9
Lambda Substitution	10
Renaming	11
Concise Pico Typechecker	13
Generating Dot files	14
Pico Typecheck using dynamically scoped variables	15
Pico eval with dynamically scoped variables	16
Pico control flow extraction	17
Pico use def extraction	18
Pico uninitialized variables	19
Pico common subexpression elimination	19
Pico constant propagation	20
Syntax Definition	21
Integration with Tscripts (Outdated)	21
Introduction	21
Requirements	21
Different styles of Type Declarations	21
Global Flow of Control	22
Modularization	23
Prototyping/implementation of Rascal	23
Issues	24

Note

This document is a braindump of ideas. See the section called “*Issues*” [] 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In the section called “*Integration with Tscripts (Outdated)*”[21] we will also explore the issues when we take integration one step further and also include Tscripts in the considerations.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-polynomial 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 = Prefix 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 can be summarized as follows:

- "Syntax modules" are identical to standard SDF modules and define concrete syntax.
- Ordinary modules define types and functions.
- 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.
- Expressions correspond roughly to Rscript expressions with some extensions:
 - There is a notation of "pattern": a quoted concrete syntax fragment that may contain variables and subexpressions to be evaluated.
 - There are lists, sets and relations together with comprehensions for these types.
 - A "select" expression corresponds to the matching provided by a set of rewrite rules. However, it provides only matching at the top level of 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 as long as replacements are possible.
 - A "search" expression also corresponds to a traversal function. Given a term and a list of patterns, it returns a list all subterms that match any of patterns.
 - Generators in comprehensions may range over syntax trees.
 - Replace, search and generators have a strategy option to indicate:
 - all = continue
 - first = break
 - td = top-down
 - bu = bottom-up
- A function contains a number of variable declarations followed by an expression.
- There is a notion of global (dynamically scoped) variables. Status not yet clear.

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

Examples

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

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`.

```
syntax module Booleans

exports
  sorts Bool

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

module Bool-examples
imports Booleans

fun Bool and(Bool B1, B2) { %% reduce & operator
  Bool B;
  select B1 {
    [| true |] => reduce(B2)
    [| false |] => [| false |]
    [| $B |]   => [| $B1 & $B2 |]
  }
}

fun Bool or(Bool B1, B2) { %% reduce | operator
  Bool B;
  select B1 {
    [| true |] => [| true |]
    [| false |] => reduce(B2)
    [| $B |]   => [| $B1 | $B2 |]
  }
}

fun Bool reduce(Bool B){
  Bool B1, B2, B3;
  select B {
    [| $B1 & $B2 |] => and(B1, B2)
    [| $B1 \ | $B2 |] => or(B1, B2) %% Note: | has to be escaped!
    [| $B3 |]       => B
  }
}
```

Deep tree matching instead of tree traversal

```
%% Mark a pure syntax module as such with "syntax"

syntax module Tree-syntax
imports Naturals
exports TREE
  context-free syntax
    NAT          -> 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: NAT N : T generates alle NAT leaves in the tree
%% # is the built-in length-of operator

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

%% Ex1b: an equivalent, more purist, version of the same function:
fun int cnt(TREE T) {
  #{N | [| $(NAT 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 "NAT"

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) {
  int N;
  replace T {
    [| $N |] => [| $(N + 1) |]
  }
}

%% Ex4: full replacement of g by i
%% Question: how about the traversal order?
%% Probable answer: bottom-up and top-down are both ok so it does
%% not matter in this example.
%% Bottom-up case: g's are replaced by i's.
%% Top-down case: g's are replaced by i's but in the arguments of
%% the new i-tree g's are later replaced by i's.
%% Question: Is it relevant to fix the traversal order?
%% The choice should then be bottom-up since top-down can be
%% achieved with recursion (see srepl, below).

fun TREE frepl(TREE T) {
  TREE T1, T2;
  replace T {
    [| g($T1, $T2) |] => [| i($T1, $T2) |]
  }
}

```

```

}

%% Ex5a: deep replacement of g by i which usually requires
%% a bottom-up traversal + break;

fun TREE drepl(TREE T) {
  TREE T1, T11, T12, T2, T21, T22;
  replace T {
    [| g($T1, $T2) |] => [| i($T1, $T2) |]
    when not exists [| g($T11, $T12) |] : T1,
    not exists [| g($T21, $T22) |] : T2
  }
}

%% Assume that the traversal is always bottom-up
%% and introduce a "first" directive (= break).
%% Along this line "all" would be the default.

fun TREE drepl(TREE T) {
  TREE T1, T2;
  replace T first {
    [| g($T1, $T2) |] => [| i($T1, $T2) |]
  }
}

%% If we want the whole repertoire of traversal functions
%% can be made available:
%% - replace T first bu { ... }
%% - replace T all td { ... }
%% - etc.
%% A nice touch is that these properties are not tied to the
%% declaration of a traversal function (as in ASF+SDF) but to
%% its use. The above would become:

fun TREE drepl(TREE T) {
  TREE T1, T2;
  replace T first bu {
    [| g($T1, $T2) |] => [| i($T1, $T2) |]
  }
}

%% Ex6a: shallow replacement of g by i (i.e. only outermost
%% g's are replaced); Usually done with a top-down traversal.
%% For brevity, we use here a string variable FUN that matches
%% a function symbol: do we want this? It corresponds to a
%% lexical variable.

fun TREE srepl(TREE T) {
  TREE T1, T2;
  str FUN;

  select (T) {
    [| g($T1, $T2) |] => [| i($T1, $T2) |]
    [| $FUN($T1, $T2) |] => [| $FUN($(srepl(T1)) $(srepl(T2))) |]
  }
}

%% Using traversal attributes, one could write this as:

```

```
fun TREE srepl(TREE T) {
  TREE T1, T2;
  replace T first td {
    [| g($T1, $T2) |] => [| i($T1, $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) {
  TREE T1, T2;
  { S | first td STATEMENT S : T, [| g($T1, $T2) |] == S }
}

or maybe abbreviated as (eliminating the need for a variable S):

fun set[TREE] find_outer_gs(TREE T) {
  TREE T1, T2;
  search T first td {
    [| g($T1, $T2) |]
  }
}

and similar for inner g's:

fun set[TREE] find_inner_gs(TREE T) {
  TREE T1, T2;
  search T first bu {
    [| g($T1, $T2) |]
  }
}
```

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

fun Bool reduce(Bool B) {
  Bool B1, B2;
  replace B bu {
    [| true & $B2 |]   => B2
    [| false & $B2 |]  => [| false |]

    [| true \| $B2 |]  => [| true |]
    [| false \| $B2 |] => B2
  }
}
```

Dot Product

[Example taken from TXL documentation]


```
syntax module Vectors
sorts Vector

context-free syntax
  "(" NAT* ")"      -> Vector

  Vector "." Vector -> NAT %% innerproduct, e.g. (1 2 3).(3 2 1) => 10

module Innerproduct

imports Vectors

fun int innerProduct(Vector V1, V2){
  NAT N1, N2;
  NAT* Rest1, Rest2;

  if [| ($N1 $Rest1) |] := V1 &
    [| ($N2 $Rest2) |] := V2
  then
    (N1*N2) + innerProduct([| ($Rest1) |], [| ($Rest2) |])
  else
    0
  fi
}
```

Bubble sort

[Example taken from TXL documentation]

```
module Bubble

fun NAT* sort(NAT* Numbers){
  NAT N1, N2;
  NAT* Rest1, Rest2;
  replace Numbers {
    [| $Rest1 $N1 $N2 $Rest2 |] => [| $Rest1 $N2 $N1 $Rest2 |]
    when N1 > N2
  }
}
```

Are \$Rest1 and \$Rest2 needed? Would

```
replace Numbers {
  [| $N1 $N2 |] => [| $N2 $N1 |]
  when N1 > N2
}
```

be ok?

Generic Bubble sort

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

```
module Bubble

fun &ELEM* sort(&ELEM* Elements, fun bool GreaterThan(&ELEM, &ELEM)){
  &ELEM E1, NE2;
  &ELEM* Rest1, Rest2;
  replace Elements {
```

```
    [| $Rest1 $E1 $E2 $Rest2 |] => [| $Rest1 $E2 $E1 $Rest2 |]  
    when GreaterThan(E1, E2)  
  }  
}
```

Do we want this generality? What are the implications for the implementation?

Lambda Substitution

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.

```
syntax module Lambda  
  
sorts Var %% variables  
      Exp %% expressions  
  
context-free functions  
  "prime" "(" Var ")" -> Var  %% generate unique name  
  Var      -> Exp  %% single variable  
  "fn" Var ">" Exp -> Exp  %% function abstraction  
  Exp Exp      -> Exp  %% function application  
  
%%-----  
  
module Lambda-Examples  
imports Lambda  
  
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) {  
  Var V2;  
  Exp Ea, Eb;  
  
  select E2 {  
    [| $V2 |] => V2  
    when V1 != V2
```

```

[| $V2 |] => E1
  when V1 == V2

[| $Ea $Eb |] => [| $(subst(V, E, Ea)) $(subst(V, E, Eb)) |]

[| fn $V2 => $Ea |] => [| fn $V2 => $Ea |]
  when V1 == V2

[| fn $V2 => $Ea |] => [| fn $V2 => $(subst(V1, E1, Ea)) |]
  when V1 != V2,
    not(V1 in freeVars(E2) & V2 in freeVars(E1))

[| fn $V2 => $Ea |] =>
[| fn $V3 => $(subst(V1, E1, subst(V2, V3, E2))) |]
  when V1 != V2,
    V1 in freeVars(Ea) & V2 in freeVars(E1),
    V3 := fresh(V2, freeVars(Ea) union freeVars(E1))
}

```

Renaming

```

syntax module Let
sorts Var %% variables
      Exp %% expressions
context-free functions

      Var                                     -> Exp %% single variable

      "let" Var "=" Exp "in Exp" "end" -> Exp %% function abstraction
%%-----

module Let-Example
import 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) {
  int Cnt1;
  Var V, Y;
  Exp E1, E2;

  select(E) {
    [| let $V = $E1 in $E2 end |] =>
      [| let $Y = $(rename(E1, Rn, Cnt))
        in
          $(rename(E2, {<V, Y>} union Rn, Cnt1))
        end
      |]
    when Y := "x" + Cnt,      %% this + operator concatenates
                              %% (after converting the int to str)
      Cnt1 := Cnt + 1

    [| $V' |] => V1
  }
}

```

```
        when { V1 } == Rn[V]

        [| $E' |] => E
    }
}

%% 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;

    select 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;

    select 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]

var
    DECLS DeclS;
    {STATEMENT ";" } * Series;
    Id Id;
    TYPE Type;
    Env Env;

fun list[Error] tcp(PROGRAM P) {
    select P {
        [| begin $DeclS $Series end |] =>
            [ tcst(S, Env) | Stat S : Series ] %% list comprehension
            when Env := {<Id, Type> |
                [| $(PICO-ID Id) : $(TYPE Type) |] : DeclS}
    }
}

fun list[Error] tcst(Stat Stat, Env Env) {
    select Stat {
        [| $Id := $Exp |] => type-of(Exp, Type, Env)
            when {<Id,Type>} := Env[Id]

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

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

```
}

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

  select E {
    [| $NatCon |] => [|
      when Type == natural

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

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

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

%% Discussion: it may be interesting to generalize select to allow
%% multiple arguments:
select (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.
```

Generating Dot files

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 |}]
}
```

Pico Typecheck using dynamically scoped variables

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 := {};
  select 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;
  select (Stats) {
    [| |] => true

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

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

    [| if $Exp then $Stats1 else $Stats2 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;

```

```
select (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

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;

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

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

fun VEnv evalIdTypes(Id-Type* idtypes) {
  Id id;
  Id-Type* tail;
  select 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;
  select series {
    [| $stat; $stats |] => venv
      when venv := evalStatement(stat),
            venv := evalStatements(stats)

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



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

Pico control flow extraction

```
module Pico-controlflow  
imports pico/syntax/Pico  
  
type CP = EXP | STATEMENT;      %% A Code Point, union of two types
```

```

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

fun cf_segment cflow({STATEMENT ";"* Stats){
  Stat Stat;
  Stat+ Stats2;
  rel[CP,CP] R1, R2;
  set[CP] En1, En2, Ex1, Ex2;

  select Stats {
    [| Stat ; $Stats2 |] =>
      <En1, R1 union R2 union (Ex1 x En2), Ex2>
      when <En1, R1, Ex1> := cflow(Stat),
           <En2, R2, Ex2> := cflow(Stats2)

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

fun cf_segment cflow(STATEMENT Stat){
  Exp Exp;
  Stat* Stats, Stats1, Stats2;
  rel[CP,CP] R1, R2;
  set[CP] En1, En2, Ex1, Ex2;

  select Stat {
    [| while $Exp do $Stats od |] =>
      <{Exp}, ({Exp} x En) union R union (Ex x {Exp}), {Exp}>
      when <En, R, Ex> := cflow(Stats)

    [| if $Exp then $Stats1 else $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)

    [| $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);
  cf_segment 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 subexpression elimination

```

module Pico-common-subexpression

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 {
    [| 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 first 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:

```
cse(P, [| $(PICO-ID Id) := $(EXP E) |])
```

or

```
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

fun Boolean is-constant(EXP E) {
  Int Int;
  Str Str;
  select (E){
    [| $Int |] => true

    [| $Str |] => true

    [| $E |] => false
  }
}

fun PROGRAM cp(PROGRAM P) {
  rel[STATEMENT, PICO-ID] Defs := defs(P);
  rel[CP,CP] Pred := cflow(P).graph;
  PICO-ID Id, Id1, Id2;

  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 {  
        [| $Id |] => E  
        when { 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.

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 amalgam 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 execution 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.

Prototyping/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 reimplementing of matching & rewriting in ASF+SDF.
 - Bound to be very slow.
 - Effort: 4 weeks
- Implementation of a typechecker in Rascal.
 - Interesting exercise to assess 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 option. 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 reimplementing of matching & rewriting in Java.
 - Effort: 8 weeks.

Issues

- How will we quote patterns and terms? Currently we use the convention:
 - Pattern is enclosed between `[]` and `[]`

- Variables are prefixed with \$
- A declared variable is written as \$(Type Var).
- An expression E to be computed and inserted is written as $$(E)$.
- Examples:

```
[ | while $Exp do $Stats od | ]
```

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

```
[ | let $Y = $(rename(E1)) in $(rename(E2)) end | ]
```

This is good for disambiguation but gives notational overhead. Is there a more concise scheme?

- More questions about patterns:
 - Do we want the subexpressions in patterns?
 - Should "patterns" be first-class citizens that can be used as values?
 - Do we want string variables in patterns?
 - 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?
- The relation (no pun intended) between local variable declarations in functions, patterns and comprehensions has to be established.
- Do we add column names in relations (as Jurgen and Bob are currently doing)? Currently both are supported.
- 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 select 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.
- Should "functions" be first-class citizens that can be used as values?
- Shopping list of ideas in Tom:
 - Named patterns to avoid building a term, i.e. $w@[| \text{while } \$Exp \text{ do } \$stat \text{ od } |]$.
 - Anti-patterns, i.e. the complement of a patterns: $! [| \text{while } \$Exp \text{ do } \$stat \text{ od } |]$ 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.