Rascal Requirements and Design Document

Paul Klint
Tijs van der Storm
Jurgen Vinju

Table of Contents

Note

This document is a braindump of ideas that is slowly converging to a coherent design. See the section called “*Issues*” [30] for the issues that have to be resolved.

Rationale

In the domain of software analysis and transformation, there exists a phletora of domain-specific languages for defining grammars, rewrite rules, software analysis and the like. So why embark on the design of yet another DSL in this area? We see the following arguments for this:

- Many existing DSLs are based on more or less exotic concepts that do excite researchers but are frightening for users without an appropriate research background.
- The notation used in many DSLs is uninviting to say the least.
- The scope of DSLs is usually narrow (this is where the word "domain-specific" kicks in). The tasks involved in carrying out a code analysis or renovation project require several DSLs. Integration between these DSLs is insufficient.
- As designers of various DSLs (ASF+SDF, Tscript, Rscript, ...) we have seen the positive as well as the negative side of DSLs. We certainly know howto implement DSLs in this area.
- We see an opportunity for a user-friendly, conceptually high-level, and rich DSL for all tasks related to software analysis and transformation.

With this background and an essential dose of optimism, we embark on this trip ... .

Introduction

The goals of the envisaged language (with working name Rascal) 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.
- Unsuprising syntax and concepts. Where possible we will stay close to C and Java notation.

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. This basically means that we include most features of the RScript language into ASF+SDF. Although we take these languages as conceptual starting point, Rascal is a completely new design.

In the section called "*Integration with Tscripts (Outdated)*" [32] 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: Old ASF+SDF programs are translatable to Rascal.
- R3: Parsing and compilation speed needs to be faster than ASF+SDF, comparable to the speed of the Java compiler. Parsetable generation is a major bottleneck in current ASF+SDF.
- R4: Concrete syntax: for readability and easy parsing of a wide range of source languages.
- R5: (**dropped**) Functional (no side-effects), dropped due to incompatibility with R7, R6
- 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: Try to keep features orthogonal: try to keep the number of ways to write down a program minimal.
- R12 (**new**): Minimize possible syntactic ambiguities; resolve them by type inference.
- R13 (**new**): Can integrate with Eclipse analysis and refactoring infra-structure, such as providing synopsis and previews of source-to-source transformations.

Rascal at a glance

Rascal consists of the following elements:

- Modules to group definitions.
- A type system and corresponding values.
- Variables to associate a name with a value in some scope.
- Parameterized functions.
- Abstract and concrete patterns to deconstruct (match) values and to construct (make) them.
- Expressions provide the elementary computations on values.
- Statements for providing more advanced control flow in computations.

These elements are summarized in the following subsections.

Modules

Modules are the organizational unit of Rascal. They may:

- import other modules (either Rascal modules or SDF modules).
- Define datatypes, subtypes or functions.
- Be parameterized with the names of formal types that are instantiated with an actual type when the module is imported.

Types and Values

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

- Symbols (as defined by an SDF 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).
- There is a catch-all type "Value".
- Relations and tuples can have optional column names.
- There is subtyping (as opposed to aliasing of types in Rscript)
- All syntactic types are a subtype of the type TREE that corresponds to AsFix. Up casts from a subtype to an enclosing type are automatic; Down casts require a run-time check.
- Datatype declarations may introduce new structured types.
- Types may include type variables like `&T` as in Rscript.

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

The root of the type system is the type Value. It has the following subtypes:

- bool
- int
- str
- loc
- void
- lists, sets, tuples, and relations of Values.
- datatype: all structures defined with a datatype definition. One of the subtypes of datatype is Tree.
  - Tree is a subtype of datatype and is the type that corresponds with concrete syntax trees (AsFix in the case of the Meta-Environment). Tree has as subtype all types that are derived from SDF definitions, i.e., all notions that are defined using a grammar.
  - The type Tree is "special" in the following sense:
    - Parsers generates values of type Tree.
    - Although the datatype Tree can be defined in Rascal, its definition is built-in in order to preserve the consistency with the parser.

Note

There is inconsistency in naming here. Probably switch to "tree", "value", etc. so that all built-in types are in lowercase.

Variables

Variables are names that have an associated scope and in that scope they have a value. A variable declaration consists of a type followed by the variable name and---depending on the syntactic position---they are followed by an initialization. Variables may be introduced at the following syntactic positions:

- As formal parameters of a function. Their scope is the function and they get their initial value when the function is called.
- Local variables in a function body are declared and initialized. Their scope is the function body.
- Variables in patterns. For patterns in match positions, variables are initialized during the match and their scope is the rule in which they occur. For patterns in make positions, the values of variables are taken from the local scope.
- Variables that are introduced by generators in comprehensions or for statement, have the comprehension, respectively, for statement as scope.
- Global variables are declared and optionally initialized at the top level of each module. Functions that use a global variable have to be explicitly declared it as well. The value of a global variable can be used and replaced by all functions that have locally declared it.

We will see below that there are certain contexts in which assignments to variables are undone in the case of failure.

Functions

A function declaration consists of a visibility declaration, the keyword `fun`, result type, a function name, typed arguments and a function body. Function without a result type have type `void`.

A visibility declaration is one of the keywords `public` or `private` (default).

A function body contains optional variable declarations (with optional initializations) followed by a number of statements. A return statement produces the value of the function.

Functions can later be extended:

Note

Function extension is still under discussion.

- The keyword `extend` before a function declaration extends a previously defined function with the same signature.
- Local declarations in the extension function are added to the original function.
- If both bodies consist of a `switch` or `visit` construct, the cases are merged.
- Other cases may be considered: switch + expr, expr + switch, visit + expr, and expr + visit.
- No other extensions are allowed.

Patterns

We distinguish two kinds of patterns:

- *Abstract* patterns: prefix dataterms that are generated by a signature.

- *Concrete* patterns: textual fragment that are generated by a context-free grammar.

Patterns may contain variables and can occur in two syntactic positions:

- *Match* positions where the patterns is matched against another term and the variables in the pattern are bound when the match is successfull. Examples of match positions are:
  - After a `case` keyword in `switch` or `visit` statement.
  - In a generator, where generated values are matched against the pattern.
- *Make* positions where the pattern is used to construct a new term (after replacing any variables in the pattern by their values. Examples of make positions are:
  - If the pattern is preceeded by the keyword `make`.

Abstract Patterns

Datatype declarations introduce a signature of abstract terms.These terms (possibly including typed variables as introduced for concrete patterns) may be used as abstract patterns at the same position where concrete patterns are allowed. Subtype declarations define an inclusion relation between types.

Concrete Patterns

There is a notation of a *concrete pattern*: a (possibly quoted) concrete syntax fragment that may contain variables. 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 quoted 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.
- Generators in comprehensions may range over syntax trees.
- Generators may have a strategy option to indicate:
  - all = continue
  - first = break
  - td = top-down
  - bu = bottom-up
- The complete repertoire of operators in Rscript is available but we have applied some rationalization:
  - + is now used for addition, union and concatenation (union is gone).
  - - is now used for substraction, difference (\ operator is gone)
  - * is now used for multiplication and intersection (inter is gone).
  - There are new assignment operators +=, -=, *=.
  - For a binary relation R that is a map (i.e. it associates a single value with each domain element) R(n), returns the single image value corresponding with n.
  - For an arbitrary binary relation, R{n} returns a set of corresponding image elements.
  - The Rscript image notation R[n] and R[-,n] are discontinued.

Statements

Rascal has the following statement types:

- Variable declaration with initialization.
- An assignment statement assigns a value to a (local or global) variable.
- If-then statement and if-then-else statement.
- A "return" statement returns a value from a function, or just returns (for functions with result type void).
- A "yield" statement that delivers a replacement value during a traversal initiated by a visit statement.
- A "switch" statement is similar to a switch statement in C or Java and for a given subject term, it 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 the subject term and does not traverse it. The type of each pattern must be identical to the type of the subject term (or be a subtype of it). It is an error if no case matches.
- A rewrite rule consists of a Pattern followed by : and a statement that returns the replacement value:

```
case Pattern : Statement
```

A statement may consist of declarations and statements. To maintain some resemblance with rewrite rules, we also support the form

```
case Pattern => Replacement
```

which is an abbreviation for

```
case Pattern : yield Replacement
```

- A "visit" statement corresponds to a traversal function. Given a subject term and a list of rewrite rules it traverses the term. Depending on the precise rules it may perform replacement (mimicking a transformer), update local variables (mimicking an accumulator) or a combination of these two. The visit statement may contain the same strategy options as a generator and also:
  - repeat = compute a fixed-point: repeat as long as the traversal function changes values.
- A "for" statement to repeat a block of code.
- A "solve" statement to solve a set of linear equations.

Failure and side-effects

There are two contexts in which side-effects, i.e., assignment to variables, have to be undone in case of failure. These contexts are a rewrite rule in a switch or visit statement. If the pattern on the left-handside of the rule matches there are various possibilities:

- All control flow path through the right-hand side of the rule end in a return statement. In this case, the rule can not fail and all side-effects caused by the execution of the right-hand side are committed.
- One or more control path can fail. This can be caused by an explicit fail statement or an if-then statement with missing else-branch. In the case of failure all side-effects are undone.

  ### Note

  Will we undo all side-effects or only the side-effects on global variables?

- If a rule fails there are two possibilities:
  - the left-hand side contains a list pattern that has more matching options; the next option is tried.
  - the left-hand side contains a list patterns that has no more matching options or it contains no list pattern at all; the next rule is tried.

Examples

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

Booleans

It seems that every language specification effort has to produce a specification of the Booleans at some moment, so let's try it now. We try the following versions:

- A version using visit, see the section called “*Booleans using visit*” [9].
- A version using an implicit reduction function

We use the following common syntax:

```
module Booleans-syntax
exports
  sorts Bool

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

Booleans using visit

A simple solution exists using the visit construct that we have encountered in the above examples.

```
module Bool-example1
imports Booleans-syntax;

fun Bool reduce(Bool B) {
    visit bu B {
      case true & <Bool B2>  => B2    %% Style 1: Use Variables
      case false & <Bool B2> => false

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

Observe that there are two styles:

- Using variables on the left-hand side: the visit is needed to fully normalize the result.
- A truth table: this is sufficient as is.

Abstract Booleans

In the above example we used concrete syntax for Booleans expressions. It also possible to define Booleans as abstract terms.

```
module Bool-abstract

datatype Bool true;
datatype Bool false;
datatype Bool and(Bool,Bool);
datatype Bool or(Bool, Bool);

fun Bool reduce(Bool B) {
    visit bu B {
      case and(true, Bool B2) => B2    %% Style 1: Use Variables
      case and(false, Bool)   => false

      case or(true, true)     => true  %% Style 2: Use a truth table
      case or(true, false)    => true
      case or(false, true)    => true
      case or(false, false)   => false
    }
```

```
}
```

First, type declarations are used to define the abstract syntax of the type Bool. Next, a similar reduce function is defined as before, but now we use abstract patterns.

Booleans with implicit reduce (NOT YET DECIDED)

In ASF, values are always reduced to a normal form when they are created. For some applications this normalization or canonicalization feature is very handy. We introduce the following syntax, which can also help in the transformation of old ASF+SDF programs to Rascal:

```
rules (Bool) {
  and(true, <Bool B2>) => B2
  and(false, Bool) => false
}
%% or
rules (Bool) {
  and(true, <Bool B2>) : { yield B2 }
  and(false, Bool) : { yield false }
}
```

These rules are applied on every Bool that is constructed. Like in ASF+SDF it is the responsibility of the programmer to make sure the rules are confluent and terminating. The body of a rules definition has the same syntax and semantics as the switch construct, allowing backtracking, side-effects and checking of conditions.

There are some issues here:

- It should be dissallowed to have private rules on public constructors; normalization is a global effect on public data-structures. On the other hand, constructors that are local to a module may have some private rules applied to them; but public rules on private constructors are dissallowed too.
- We previously had other ideas about this feature. We introduced a reduce keyword that would call a certain named function at the construction of every Bool. The problem with that feature is that the operational semantics (in the eyes of the Rascal programmer) really looks like first some term is to be created, and then a function will be applied to normalize it. In reality though, we want to normalize the terms in an innermost fashion, and at construction time, so that the terms on the left-hand side of the rules are never actually created or allocated.

Abstract Graph datatype

In the Meta-Environment we use an abstract data type to exchange data representing graphs. It can be defined as follows.

```
module Graph

datatype Graph graph(NodeList nodes,
                     EdgeList edges,
                     AttributeList attributes);

subtype Nodelist list[Node];

datatype Node node(NodeId id,
                   AttributeList attributes);

datatype Node subgraph(NodeId id,
                   NodeList nodes,
                   EdgeList edges,
```

```
                   AttributeList attributes);

datatype NodeId id(term id);

subtype AttributeList list[Attribute];

datatype Attribute bounding-box(Point first, Point second);
datatype Attribute color(Color color);
datatype Attribute curve-points(Polygon points);
datatype Attribute direction(Direction direction);
datatype Attribute fill-color(Color color);
datatype Attribute info(str key, term value);
datatype Attribute label(str label);
datatype Attribute tooltip(str tooltip);
datatype Attribute location(int x, int y);
datatype Attribute shape(Shape shape);
datatype Attribute size(int width, int height);
datatype Attribute style(Style style);
datatype Attribute level(str level);
datatype Attribute file(File file);
datatype Attribute file(term file);

datatype Color rgb(int red, int green, int blue);

datatype Style bold | dashed | dotted |
               filled | invisible | solid;

datatype Shape box | circle | diamond | egg |
               elipse | hexagon | house | octagon |
               parallelogram | plaintext | trapezium |
               triangle;

datatype Direction forward | back | both | none;

subtype Edgelist list[Edge];

datatype Edge edge(NodeId from,
                   NodeId to,
                   AttributeList attributes);

subtype Polygon list[Point];

datatype Point point(int x, int y);
```

Tree traversal

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

```
module BTree-syntax
imports basic/Integers

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

```
    i(BTREE,BTREE)  -> BTREE
```

```
module BTree-Examples
imports BTree-syntax;

%% Ex1: Count leaves in a BTREE
%% Idea: int N : T generates alle Integer leaves in the tree
%% # is the built-in length-of operator

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

%% Ex1: an equivalent,  more purist, version of the same function:
fun int cnt(BTREE T) {
    return #{N | <Integer N> : T}
}

%% Ex1: alternative solution using trafo functions:

fun int cnt(BTREE T) {
    int C = 0;
    visit T {
      case <Integer N> : C = C+1
    };
    return C;
}

%% Ex2: Sum all leaves in a BTREE
%% 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(BTREE T) {
    return sum({N | int N : T});
}

%% Ex2: using accumulator

fun int cnt(BTREE T) {
    int C = 0;
    visit T {
      case <Integer N> : C = C+N
    };
    return C;
}

%% Ex3: Increment all leaves in a BTREE
%% Idea: using the construct "visit T { ... }" visit all leaves in the
%% tree T that match an integer and replace each N in T by N+1.
%% The expression as a whole returns the modified term.
%% 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 BTREE inc(BTREE T) {
```

```
    visit T {
      case <Integer N>: yield (N + 1)
    };
}

%% Ex4: full replacement of g by i
%% The whole repertoire of traversal functions is available:
%% - visit first bu T { ... }
%% - visit all td  T { ... }
%% - 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 BTREE frepl(BTREE T) {
    visit all bu T {
      case g(<BTREE T1>, <BTREE T2>) => make i(<BTREE T1>, <BTREE T2>)
    };
}

%% Ex5: Deep replacement of g by i

fun BTREE frepl(BTREE T) {
    visit first bu T {
      case g(<BTREE T1>, <BTREE T2>) => make i(<BTREE T1>, <BTREE T2>)
    };
}

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

fun BTREE srepl(BTREE T) {
    visit first td T {
      case g(<BTREE T1>, <BTREE T2>) => make i(<BTREE T1>, <BTREE T2>)
    };
}


%% Ex7: We can also add the first/td directives to all generators
%% (where "all td" would be the default):

fun set[BTREE] find-outer-gs(BTREE T) {
    return
    { S | STATEMENT S : first td T,
          g(<BTREE T1>, <BTREE T2>) := S };
}

%% Ex8: accumulating transformer that increments leaves with amount D and count
fun tuple(int, BTREE) count-and-inc(BTREE T, int C, int D) {
    int C = 0;

    visit T {
      case <Integer N>: { C = C + 1; yield N+D }
     };
     return <C, T>;
```

```
}
```

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) {
    return {V | Var V : E}
}

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

fun set[Var] freeVars(Exp E) {
    return 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){ return prime(V); } else {return V;};
}

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

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

    switch E2 {
      case <Var V2>: if(V1 != V2){ yield V2; }

      case <Var V2>: if(V1 == V2){ yield E1; }

      case <Exp Ea> <Exp Eb>: {
        Exp EaS = subst(V, E, Ea);
        Exp EbS = subst(V, E, Eb);
        return make <Exp EaS> <Exp EbS>;
      }
```

```
      case fn <Var V2> => <Var Ea>:
        if (V1 == V2) { yield make fn <Var V2> => <Exp Ea> }

      case fn <Var V2> => <Exp Ea>:
        if(V1 != V2 and not(V1 in freeVars(E2) and
           V2 in freeVars(E1))){
           Exp E1S = subst(V1, E1, Ea);
           yield make fn <Var V2> => <Exp E1S>;
        }

      case fn <Var V2> => <Exp Ea>:
        if(V1 != V2 and V1 in freeVars(Ea) and
           V2 in freeVars(E1)){
           Var V3 = fresh(V2, freeVars(Ea) + freeVars(E1));
           Exp EaS = subst(V1, E1, subst(V2, V3, E2));
           yield make fn <Var V3> => <Exp EaS>;
        }
    };
}
```

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 {
    case let <Var V> = <Exp E1> in <Exp E2> end: {
         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>} + Rn, Cnt1);
         return make let <Var Y >= <Exp E1R>
                     in
                        <Exp E2R>
                     end;
          }
```

```
    case <Var V>: return Rn[V]

    default: return E
    };
}
```

Renaming in Let using globals

Here is the same renaming function now using two global variables.

```
module Let-Example
imports Let;

%% Rename all bound variables in an Exp
%% Version 2: using global variables
%% Cnt: global counter to generate fresh variables
%% rel[Var,Var]: global renaming table

global int Cnt = 0;
global rel[Var,Var] Rn = {};

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

fun Exp rename(Exp E) {
    global int Cnt;
    global rel[Var,Var] Rn;
    switch E {
    case let <Var V> = <Exp E1> in <Exp E2> end: {
         Var Y = newVar();
         Rn = {<V, Y>} + Rn;
         Exp E1R = rename(E1);
         Exp E2R = rename(E2);
         return make let <Var Y >= <Exp E1R>
                     in
                        <Exp E2R>
                     end;
          }

    case <Var V>: return Rn(V)

    default: return E
    };
}
```

Concise Pico Typechecker

The following example shows the tight integration ASF with comprehensions.

```
module Typecheck

imports Pico-syntax;
imports Errors;

subtype Env rel[PICO-ID,TYPE];
```

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

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

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

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

fun list[Error] type-of(Exp E, TYPE Type, Env Env) {
    switch E {
      case <NatCon N>: if(Type == natural){ return []; }

      case <StrCon S>: if(Type == string) { return []; }

      case <PICO-ID Id>: {
         TYPE Type2 = Env(Id);
         if(Type2 == Type) { return []; }
      }

      case <EXP E1> + <EXP E2>:
        if(Type == natural){
           return type-of(E1, natural, Env) +
                  type-of(E1, natural, Env);
        }

      case <EXP E1> + <EXP E2>:
        if(Type == natural){
           return type-of(E1, natural, Env) +
                  type-of(E1, natural, Env);
        }

      case <EXP E1> || <EXP E2>:
        if(Type == string){
          return type-of(E1, string, Env) +
                 type-of(E1, string, Env)
        }

      default: return [error("Incorrect type")]
    };
```

```
}
```

Pico evaluator

```
module Pico-eval
imports pico/syntax/Pico;

subtype PICO-VALUE int;
subtype PICO-VALUE str;

subtype VEnv rel[PICO-ID, PICO-VALUE];

fun VEnv evalProgram(PROGRAM P){
    switch P {
      case begin <DECLS Decls> <{STATEMENT ";"}* Series> end: {
          VEnv Env = evalDecls(Decls);
          return evalStatements(Series, Env);
      }
   }
};

fun VEnv assign(VEnv Env, PICO-ID Id, PICO-VALUE V){
  return Env +> {<Id, V>}  %% we need a nice tuple replacement operator here
}

fun PICO-VALUE valueOf(VEnv Env, PICO-ID Id){
  return Env(Id);
}

fun VEnv evalDecls(DECLS Decls){
    VEnv Env = {};
    visit Decls {
      case <PICO-ID Id> : string:  Env = assign(Env, Id, "")
      case <PICO-ID Id> : natural: Env = assign(Env, Id, 0)
    }
}

fun VEnv evalStatements({STATEMENT ";"}* Series, VEnv Env){
    switch Series {
      case <STATEMENT Stat>; <{STATEMENT ";"}* Series2>: {
        Env Env2 = evalStatement(Stat, Env);
        return evalStatements(Series2, Env2);
      }
      case [| |]: return Env
    }
}

fun VEnv evalStatement(STATEMENT Stat, VEnv Env){
    switch Stat {
      case [| <PICO-ID Id> = <EXP Exp> |]: {
        PICO-VALUE Val = evalExp(Exp, Env);
        return assign(Env, Id, Val)
      }

      case if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                    else <{STATEMENT ";"}* Stats1> fi:{
        PICO-VALUE Val = evalExp(Exp, Env);
        if(Val != 0)
```

```
          return evalStatments(Stats1, Env)
        else
          return evalStatements(Stats2, Env)

      }

      case while <EXP Exp> do <{STATEMENT ";"}* Stats1> od:{
        PICO-VALUE Val = evalExp(Exp, Env);
        if(Val != 0)
          return Env
        else {
          VEnv Env2 = evalStatements(Stats1, Env);
          return evalStatement(Stat, Env2)
        }
      }
    };
};

fun PICO-VALUE evalExp(Exp exp, VEnv Env) {
    switch exp {
      case <NatCon N>: return N

      case <StrCon S>: return S

      case <PICO-ID Id>: return valueOf(Env, Id)

      case <EXP exp1> + <EXP exp2>: {
           Natural nat1 = evalExp(exp1, Env);
           Natural nat2 = evalExp(exp2, Env);
           return nat1 + nat2;
      }
      case <EXP exp1> - <EXP exp2>: {
           Natural nat1 = evalExp(exp1, Env);
           Natural nat2 = evalExp(exp2, Env);
           return nat1 + nat2;
      }
      case <EXP exp1> || <EXP exp2>: {
           StrCon str1 = evalExp(exp1, Env);
           StrCon str2 = evalExp(exp2, Env);
           return concat(str1, str2);
      }
   }
}
```

Pico evaluator with globals

Here is the same evaluator but now using a global variable to represent the value environment.

```
module Pico-eval
imports pico/syntax/Pico;

subtype PICO-VALUE int;
subtype PICO-VALUE str;

subtype VEnv rel[PICO-ID, PICO-VALUE];

fun void evalProgram(PROGRAM P){
    switch P {
```

```
      case begin <DECLS Decls> <{STATEMENT ";"}* Series> end: {
          evalDecls(Decls);
          evalStatements(Series);
      }
   }
};

fun void assign(PICO-ID Id, PICO-VALUE V){
  global Venv Env;
 Env = Env +> {<Id, V>}
  return;
}

fun PICO-VALUE valueOf(PICO-ID Id){
  global Venv Env;
  return Env(Id);
}

fun VEnv evalDecls(DECLS Decls){
    global VEnv Env = {};
    visit Decls {
      case <PICO-ID Id> : string:  assign(Id, "")
      case <PICO-ID Id> : natural: assign(Id, 0)
    }
}

fun void evalStatements({STATEMENT ";"}* Series){
    global VEnv Env;
    switch Series {
      case <STATEMENT Stat>; <{STATEMENT ";"}* Series2>: {
        evalStatement(Stat);
        evalStatements(Series2);
        return
      }
      case [| |]: return
    }
}

fun void evalStatement(STATEMENT Stat){
    global VEnv Env;
    switch Stat {
      case [| <PICO-ID Id> = <EXP Exp> |]: {
        PICO-VALUE Val = evalExp(Exp);
        assign(Id, Val);
        return
      }

      case if <EXP Exp> then <{STATEMENT ";"}* Stats1>
                   else <{STATEMENT ";"}* Stats1> fi:{
        PICO-VALUE Val = evalExp(Exp);
        if(Val != 0) {
          evalStatements(Stats1);
          return
        } else {
          evalStatements(Stats2);
          return
        }
      }
```

```
      case while <EXP Exp> do <{STATEMENT ";"}* Stats1> od:{
        PICO-VALUE Val = evalExp(Exp);
        if(Val != 0)
          return
        else {
          evalStatements(Stats1);
          evalStatement(Stat);
          return
        }
      }
    };
};

fun PICO-VALUE evalExp(Exp exp) {
    global VEnv Env;
    switch exp {
      case <NatCon N>: return N

      case <StrCon S>: return S

      case <PICO-ID Id>: return valueOf(Id)

      case <EXP exp1> + <EXP exp2>: {
           Natural nat1 = evalExp(exp1);
           Natural nat2 = evalExp(exp2);
           return nat1 + nat2;
      }
      case <EXP exp1> - <EXP exp2>: {
           Natural nat1 = evalExp(exp1);
           Natural nat2 = evalExp(exp2);
           return nat1 + nat2;
      }
      case <EXP exp1> || <EXP exp2>: {
           StrCon str1 = evalExp(exp1);
           StrCon str2 = evalExp(exp2);
           return concat(str1, str2);
      }
   }
}
```

Pico control flow extraction

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

subtype CP EXP;         %% A Code Point, union of two types
subtype CP STATEMENT;

subtype CFSEGMENT tuple(set[CP] entry, rel[CP,CP] graph, set[CP] exit);

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

```
      case [| |]: return <{}, {}, {}>
    }
};

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

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

      case [| <STATEMENT Stat> |]: return <{Stat}, {}, {Stat}>
    };
}
```

Pico use def extraction

```
module Pico-use-def

imports pico/syntax/Pico;


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

fun rel[PICO-ID, STATEMENT] defs(PROGRAM P) {
  return {<Id, S> | 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. Also note that, compared to older definitions of these functions, the iudentifier is placed as first element in each tuple.

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[PICO-ID, STATEMENT] Defs = defs(P);
    CFSEGMENT CFLOW = cflow(P);
```

```
    set[CP] Root = CFLOW.entry;
    rel[CP,CP] Pred = CFLOW.graph;

    return {Id | tuple(EXP E, PICO-ID Id) : Uses,
                 E in reachX(Root, Defs{Id}, Pred)
    };
}
```

Pico common subsexpression elimination

```
module Pico-common-subexpression

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

fun PROGRAM cse(PROGRAM P) {
    rel[PICO-ID, STATEMENT] 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)
       };

    visit P {
      case <EXP E>: if({ PICO-ID Id } := replacements{E}) yield Id
    };
}
```

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): [UNDER DISCUSSION]

```
 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 as 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 (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 {
     case <NatCon N> => true

     case <StrCon S> => true

     case <EXP E> => false
   }
}

fun PROGRAM cp(PROGRAM P) {
    rel[PICO-ID, STATEMENT] 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
      };

    visit P {
     case <PICO-ID Id>: if({ EXP E } := replacements{Id}) yield E
    };
}
```

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.

Pico Reaching definitions

Recall the equations construct as used, for example, in the reaching definitions example in the Rscript guide. It computes the values of a set of variables until none of them changes any longer. The "solve" statement achives the same effect.

```
module Pico-reaching-defs

subtype Def  tuple(Stat theStat, Var theVar);
subtype Use  tuple(Stat theStat, Var theVar);

fun set[Stat] predecessor(rel[Stat,Stat] P, Stat S) { return P[-,S] }

fun set[Stat] successor(rel[Stat,Stat] P, Stat S) { return P[S,-] }

fun rel[Stat, Def] reaching-definitions(rel[Stat,Var] DEFS,
                                        rel[Stat,Stat] PRED) {

    set[Stat] STATEMENT = carrier(PRED);

    rel[Stat,Def] DEF  = {<S,<S,V>> | <Stat S, Var V> : DEFS};

    rel[Stat,Def] KILL =
        {<S1, <S2, V>> | <Stat S1, Var V> : DEFS,
                         tuple(Stat S2, V) : DEFS,
                         S1 != S2
        };

     rel[Stat,Def] IN = {};
     rel[Stat,Def] OUT = DEF;

     solve {
           IN  =  {<S, D> | int S : STATEMENT,
                            Stat P : predecessor(PRED,S),
                            Def D : OUT{P}};
           OUT = {<S, D> |  int S : STATEMENT,
                            Def D : DEF{S} +  (IN{S} - KILL{S})}
    };
    return IN;
}
```

Symbol table with scopes

Here is a (probably naive) implementation of a symbol table that maintains a list of numbered scopes as well as a (Name, Value) mapping in each scope. Note that we introduce parameterized modules to do this right.

```
module SymTable[Name, Value]

%% A scope-oriented symbol table.
%% Each scope consists of a map from names to values.
```

```
%% THis more intended to explore whether this can be expressed
%% *at all* than that the datatype is well designed.

subtype ScopeMap rel[Name, Value];
subtype ScopeId  int
datatype STable  stable(ScopeId scope, rel[int, ScopeMap] scopes);

%% Create a new, empty,  table
fun STable new(){
  return make stable(0, {<0, {}>});
}

%% Create a new, non-empty, table
fun STable new(ScopeId scope, rel[int, ScopeMap] scopes){
  return make stable(scope, scopes);
}

%% Update, in a given scope, the value of a variable
fun STable update(STable ST, ScopeId scope, Name N, Value V){
  X = ST.scopes(scope) + {<N, V>};
  return new(scope, ST.scopes + (ST.scopes(scope) + {<N, V>}));
}

%% Get, in a given scope, the value of a variable
fun STable value(STable ST, ScopeId scope, Name N){
  ScopeMap smap = ST.scopes(scope);
  return smap(N);
}

%% update, in the current scope, the value of a variable
fun STable update(STable ST, Name N, Value V){
  ScopeId scope = ST.scope;
  ScopeMap smap = ST.scopes(scope) + {<N,V>};
  return new(scope, ST.scopes + (ST.scopes(scope) + {<scope, smap>}));
}

%% Get, in the current scope, the value of a variable
fun STable value(STable ST, Name N){
  ScopeMap smap = ST.scopes(ST.scope);
  return smap(N);
}

%% add a new scope and make it the current scope
fun STable new-scope(STable ST){
  ScopeId scope = ST.scope + 1;
  return new(scope, ST.scopes + {<scope, {}>});
}

%% switch to another scope
fun STable switch-scope(STable ST, ScopeId scope){
  return new(scope, ST.scopes);
}
```

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
       )
       return (N1*N2) + innerProduct( (<Rest1>), (<Rest2>) )
    else
       return 0;
}
```

Bubble sort

[Example taken from TXL documentation]

```
module Bubble

fun Integer* sort(Integer* Numbers){
    visit Numbers {
      case <Integer* Rest1> <Integer N1> <Integer N2> <Integer* Rest2>:
        if(N1 > N2){
          return sort(make <Integer* Rest1>
                             <Integer N2> <Integer N1>
                           <Integer* Rest>);
        }
    };
    return Numbers
}
```

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

Another way to write this is:

```
module Bubble2

fun Integer* sort(Integer* Numbers){
    visit repeat Numbers {
      case <Integer N1> <Integer N2>:
        if( N1 > N2)
            yield make <Integer N2> <Integer N1>
    };
    return Numbers
}
```

The visit will replace all adjacent pairs that are in the wrong order in the current list. This is repeated (fixed point operator) until no more changes are possible.

Generic Bubble sort [under discussion]

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

```
module Bubble-Gen

fun &Elem* sort(Elem* Elements, fun bool GreaterThan(&Elem, &Elem)){

  visit repeat Elements {
      <E1> <E2>: if(GreaterThan(E1, E2)) yield <E2> <E1>

     }
}
```

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

Read-Eval-Print Loop (REPR)

For the scripting of application it is important to have a command language and read-eval-print loop. Here is an attempt. The command prompt is ">".

```
> import lang.java.syntax.Main as Java
> str source := read("program.java");
> CU program := Java.CU.parse(source);

> accu int count(CU P, int cnt) {
>   switch P {
>     Java.Statements.IF => cnt++;
>   }
> }

> count(program)
17
```

There are several innovations here:

- The import associates a name with the imported module.

  ### Note

  This means that "grammar" and "rule" become notions that can be manipulated.

- There is a read functions that reads a text file into a string.

  ### Note

  We need an io library that reads/writes strings and data values.

- We associate a parse function with every non-terminal in a grammar.

- The notation `Java.Statements.IF` consists of three parts:

  - Language name

  - Sort name

  - Rule name (currently implemented with the "cons" attribute).

It can be used as pattern. Other potential uses are as generator:

```
{S | Java.Statements.IF S : P}
```

It generates all if statements in P.

Syntax Definition

See separate SDF definition.

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 "*Concrete Patterns*"[6] 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 |]`
- How do we identify built-in sorts (bool, int, etc) with their syntactic counterparts?
- In a list comprehension: do list values splice into the list result?
- Ditto for set comprehensions.
- Some clean up of Rscript notation (operators "o" and "x" should go).
- We need an io-library.
- We need memo functions.
- How are annotations handled?
- How is lexical matching incorporated?
- Unexplored idea: add (possibly lazy) generators for all types; this allows to generate, for instance, all statements in a program.
- 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.

Graveyard

Don't read the following sections; they are leftovers from earlier versions of this document but may still contain material that can be reused.

Mapping features to datatypes

Note

This section has played a role during initial design; it is now outdated.

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

Outdated examples

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 |}}
}
```

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.