Interner Bericht

The Refus Programming Language

Dipl.-Inform. Reinhard Eppler
Dipl.-Inform. Peter Knauber
Dipl.-Inform. Stefan Vorwieger
Prof. Dr. Hans-Wilm Wippermann

Interner Bericht 253 / 94

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-67653 Kaiserslautern

The Refus Programming Language

Dipl.-Inform. Reinhard Eppler
Dipl.-Inform. Peter Knauber
Dipl.-Inform. Stefan Vorwieger
Prof. Dr. Hans-Wilm Wippermann

Interner Bericht 253 / 94

Herausgeber: AG Programmiersprachen und Compiler, Universität Kaiserslautern
Kaiserslautern, September 1994

Contents

1. Preliminaries

This report gives a short definition of the functional programming language Refus, which was designed for development and demonstration of some new concepts. Some of them are still incomplete or inconsistent and topic of future research. This is especially the case for generic definitions and linear types which shall allow for I/O handling and in-place updates. We will not consider these concepts in this report, handling of input and output is described by a dynamic concept that shall be replaced by a static system in future releases. Refus is designed to be modular in the sense of Oberon-2. Since the available implementation provides a simple file inclusion concept only, we will not describe the module concept. Refus is characterized by the following concepts:

- static and dynamic typing
- extensible records, hierarchic type system
- best-fit pattern matching
- sets and maps
- ZF-expressions (generators) for sets, lists and maps

2. Lexical Structure

Refus programs consist of sequences of terminal symbols. Syntax rules in EBNF style define the structure of programs. Terminal symbols are identifiers, literals and operator symbols. They are separated by spaces or tabs, if necessary. Otherwise the longest sequence of characters that matches the syntax rules forms one token. Lower and upper case letters are different characters. Terminals are written in single or double quotes, alternatives are separated by "|". Optional elements are written in "[" and "]", repeated elements in "{" and "}".

Technical terms are typeset in italic and defined in appendix A. Syntax rules of the form X-list resp. X-sequ are defined as follows:

```
X-list     = X {"," X}.
X-sequ     = X {";" X} [";"].
```

2.1. Identifier

```
ident      = letter {idChar} ( {"'"} | "?" ).
idChar     = letter | digit | "_".
qualId     = ident.
```

Identifiers consist of letters, digits or "_". The first character must be a letter. An identifier may be followed by any number of single quotes "'" or a question mark. Reserved words (see 2.5) are not legal identifiers.

2.2. Numbers

```
number     = digits [ "." digits ["E" ["+" | "-"] digits]].
digits     = digit {digit}.
```

Numbers are unsigned numeric constants. Constants of type REAL must have a decimal point ".".

2.3. Strings

```
string     = """ {char} """.
```

Strings are sequences of characters and written in double primes. Every legal C-string is also a legal Refus-String.

2.4. Boolean Values

Boolean constants are *TRUE* and *FALSE*.

2.5. Reserved Words

Reserved words are:

@	#	*	+	,	-
->	.	..	/	\	|
?	!	!!	:	;	<
<-	<=	<>	=	=>	>
>=	(	)	[	]	(.
.)	(!	!)	{	}	ALL
AND	DECL	DEF	DIV	ELSE	ELSIF
END	EQUIV	EXPORT	FCT	FALSE	FI
IF	IMPL	IMPORT	IN	IN?	IS
LET	MOD	MODULE	NOT	OR	THEN
TRUE	TYPE	UNQUALIFIED		VAL	WHERE

2.6. Predefined Identifiers

Predefined identifiers are:

BOOL	DYN	INT	REAL	STRING

2.7. Separators

Separators are spaces, tabs and comments. A comment starts with "--" or "//" and ends on the same line.

3. Type Definitions

typeDef = *"TYPE" ident "=" type.*

Refus is statically typed and provides a universal type DYN. A type is a set of values. Five predefined types may be combined to new types in type expressions. Type expressions may be bound to identifiers in type definitions (*typeDef*). Such definitions are allowed to be recursive as long as no function or map types are involved. Each recursive type has a trivial value NIL. Definitions of the form *TYPE A = B* where A and B are identifiers are not permitted. Generic types will be provided in future extensions.

If *tp-id* is a type name and *constr* a constructor that defines a constant which can be *casted* to *tp-id*, then *tp-id constr* is a constructor for a value of type *tp-id*.

3.1. Basic Types

The following basic types are provided:

• INT	
• REAL	
• STRING	the set of all C strings
• BOOL	the set of boolean constants
• DYN	the set of all Refus constants

3.2. Tuples

tuple-tp = *"(" [type-list] ")".*

A tuple type is the cartesian product of the component types. Elements of such a type are called tuples. If *tp-id* is the name of some type, then *(tp-id)* is equivalent to *tp-id*.

3.3. Lists

```
list-tp      = "[" type "]".
```

A list consists of any number of elements of the same type. The list elements are ordered and indexed by their position in the list. The first element of a list has position 0. A list type is the set of all lists of the component type.

3.4. Sets

```
set-tp       = "{" type "}".
```

A set consists of any number of elements of the same type. Each value can at last once be element of a set. A set type is the powerset of the component type.

3.5. Maps

```
map-tp       = "(!" stype "->" stype "!)".
```

A map is a finite mapping. Its domain may be any finite subset of the argument type. A map type is the set of all maps of the specified domain and range type.

3.6. Records

```
rec-tp       = "(." [ super ][ decl-list ] ".)".
super        = "(" qualId-list ")".
decl         = ident-list ":" type.
```

A record defines a structure which consists of a fixed number of fields that may have different types. A record type declaration lists the names and field types of records of that type. A record type comprises all records with adequate field bindings and all elements of extensions of that type.

Named record types may be *extensions* of named record types which are called *base* types. The field definitions of the *base* types apply to the new type. Redefinition of fields is not allowed, for each field definition there must be only one type that lists that field explicitly.

3.7. Functions

```
type         = stype { "->" stype } | "FCT" stype "->" stype { "->" stype }.
stype        = qualId | tuple-tp | list-tp | set-tp | map-tp | rec-tp.
```

A function type is the set of all curried functions that map values of the specified domain to the specified range. "->" is right associative.

4. Expressions

```
expr         = boolPrim [ "EQUIV" boolPrim ].
boolPrim     = boolTerm [ "IMPL" boolTerm ].
boolTerm     = boolFactor { "OR" boolFactor }.
boolFactor   = boolOpd { "AND" boolOpd }.
boolOpd      = arithExpr [ relOp arithExpr ].
arithExpr    = term { addOp term }.
term         = factor { mulOp factor }.
factor       = [ monop ] prim { prim }.
prim         = operand { "@" operand }.
relOp        = "=" | ">" | "<" | ">=" | "<=" | "<>" | "IN?".
addOp        = "+" | "-".
mulOp        = "/" | "*" | "DIV" | "MOD".
```

monop = "+" | "-" | "*NOT*" | "#".

4.1. Operators

Expressions consist of sequences of operands that are combined by operators. Semantics and precedence of operators is listed in the following table.

Precedence	Operator	Semantics
1	EQUIV	a EQUIV b ≡ IF a THEN b ELSE NOT b FI
2	IMPL	a IMPL b ≡ IF a THEN b ELSE TRUE FI
3	OR	a OR b ≡ IF a THEN TRUE ELSE b FI
4	AND	a AND b ≡ IF a THEN b ELSE FALSE FI
5	= <> < > <= >= IN?	Boolean operations, = and <> are defined for all types. a IN? b denotes set membership. b may be a set or a list.
6	+ -	Addition resp. union, subtraction resp. set difference
7	* / MOD DIV	Multiplication resp. intersection, division resp. symmetric set difference, integer division resp. remainder
8	NOT - + #	Negation, monadic + and -, number of elements of a set or list, length of a string.
9	(no operator)	Function application

Tabelle 1: Operator Precedence

Function application is written in the form *f val*, where *f* is an expression that evaluates to a function and *val* evaluates to a value which is passed to that function. All expressions except conditional expressions and lambda expressions are eager evaluated. Function application has precedence 9.

4.2. Operands

operand = *designator* | *condition* | *literal* | *lambda*
| *tuple-constr* | *list-constr* | *set-constr* | *map-constr* | *rec-constr.*
designator = *qualId* { "." *ident* | "!" *operand* | "!!" *operand* }.
literal = *number* | *string* | "*TRUE*" | "*FALSE*".

Elementary operands are identifiers and constants of the basic types. An identifier denotes the value bound in a definition. An identifier may be succeeded by selectors for list, map and record selection. *v.fld* denotes selection of record field *fld* of record *v*, *v!i* denotes application of map *v* to *i* and *v!!i* denotes the element at position *i* in list *v*.

4.3. Constructors

tuple-constr = "(" [*expr-list*] ")".
list-constr = "[" [*elem-list* | *update maplet-list*] "]".
set-constr = "{" [*elem-list*] "}".

```
map-constr   = "(!" [ [update] [ maplet-list ] ] "!)".
rec-constr   = "(." [ [update] [ reclet-list ] ] ".)".
elem         = ".." expr | expr generators | range.
maplet       = ".." expr
             | expr "->" expr generators
             | range "->" expr.
reclet       = ".." expr | ident ":" expr.
range        = expr ".." expr.
update       = "/" designator "/".
generators   = { "|" generator }.
generator    = sGenerator { "AND" sGenerator}.
sGenerator   = "ALL" pattern "<-" arithExpr | boolOpd.
```

Constructors are used to define and modify values.

Enumeration

Lists, tuples and sets are built by enumerating their elements, maps and records by enumerating their associations. A map association is written *index -> value* which describes mapping of *index* to *value*, a record association is written *fieldId : value* which describes that component *fieldId* has value *value*. Ranges are allowed in lists and sets. Mapping of a range to some value is possible for maps. The last associations for the same *index* resp. *field* is valid, if several bindings are specified.

Example: (! 1..2->1, 2->2 !) ==> (! 1->1, 2->2 !)

Integration

The components of other data structures can be integrated in values of the same type, written as *..expr*. If a constructor is preceded by some type identifier, *expr* may denote a value of that type or a value that can be *casted* to that type.

Comprehension

Elements of the same kind can be generated by generator expressions *E*|*G*. *E* is an expression that describes the elements and *G* is a predicate that consists of generators *(ALL pattern <- expr)* and predicates, separated by operator "AND". A generator introduces new variables that are visible in *E* and *G* and describes bindings of the variables to the values of *expr*. *E* is evaluated for each variable binding, for which the predicates hold. The result becomes an element of the new data structure. In list constructors only one generator is allowed, the order of the elements of *expr* determines the order of the produced results. In map comprehensions no associations with the same index are allowed. *E* may be itself a generator expression.

Examples. let list = [1,2,3,4,5], set = {{1,2},{2,3}}

```
[ p + 1 | ALL p <- list ] ➢ [2,3,4,5,6]
[ p + 1 | ALL p <- list AND p MOD 2 = 1 ] ➢ [2,4,6]
[ p + 1 | ALL p <- list,  6 - p | ALL p <- list ] ➢ [2,3,4,5,6,5,4,3,2,1]
{ e | ALL e <- se | ALL se <- set } ➢ {1,2,3,4}
(! 0 -> x | ALL x <- X !) is only valid if #X = 1
```

Update

Records, maps and lists can be updated. Update is indicated by */designator/* where *designator* indicates which value shall be changed. Update is functional, i.e. a new value is constructed by an update constructor. For maps and records a constructor $(_x$ */designator/ assocs* $_x)$ is equivalent to $(_x$ *..designator, assocs* $_x)$ with the exception that no new indices resp. record fields must be introduced by *assocs*. New list elements are introduced by associations *position -> value* where *position* indicates which element shall be replaced.

Example: [/L/ 1..2->0] WHERE L = [1,2,3] END ➢ [1,0,0]

4.4. Conditional

condition	*= "IF" gcd { "ELSIF" gcd } "ELSE" body "FI".*
gcd	*= expr "THEN" body.*

The guards of each guarded command *gcd* are evaluated in the textual order until one holds. The corresponding *body* is used to compute the result of the conditional. If no guard holds, the *body* following the reserved word *ELSE* forms the result.

4.5. Lambda Expressions

lambda	*= ("\" | "FCT") def-pattern {def-pattern} "=>" body.*

Lambda expressions define anonymous functions. For lambda expressions the following equivalence holds: $\backslash p_1\ p_2 \ldots p_n => e$ equals $\backslash p_1 => \backslash p_2 => \ldots \backslash p_n => e$.

4.6. Scopes

body	*= ["LET" def-decl-sequ "IN"] expr ["WHERE" def-decl-sequ "END"].*
def-decl	*= def | declaration.*

New identifiers may be introduced in a *body* which is bounded by the reserved words *LET* and *END*. Each identifier is visible in the hole body which includes the expression of the body with the exception of nested bodies that redefine the identifier. Type identifiers must not be redefined. If the let-part or the where-part is missing, the body begins resp. ends with the expression of the body. Variable identifiers must not be defined recursive. Identifiers introduced in patterns belong to the scope of the associated function body.

5. Function and Variable Definition

def	*= "VAL" def-pattern "=>" body.*
	| "FCT" ident def-pattern {def-pattern} "=>" body.
declaration	*= "DECL" genDecl.*
genDecl	*= ident-list ":" type.*

Variables are introduced in VAL-definitions. Each variable of the pattern is bound to the corresponding value of the body. Pattern and body must *match*. Recursive definitions of variables are not permitted. Each variable of a recursive type may accept a value which is constructed by NIL preceded by the type name.

A function definition consists of a set of FCT-definitions which are called rules. Each rule of a function must specify the same number of *def-patterns*. Given a function application, the *most specific* rule *matching* the argument is used to compute the result of the application. There must not be *overlaps* except there are rules that cover the overlap and that are more specific than the overlapping ones. No two rules are allowed to have the same specificity.

Declarations have to be considered. The value of a declared variable must *belong* to the declared type, function arguments and results must belong to the declared types. Undeclared identifiers may accept any value.

6. Patterns

pattern	*= tp-id var-id ":" s-pattern*
	| tp-id var-id

```
                 | tp-id s-pattern
                 | var-id ":" s-pattern
                 | var-id ":" tp-id
                 | var-id
                 | s-pattern.
s-pattern        = tup-pat | list-pat | rec-pat | literal | "?".
tup-pat          = "(" [ pattern "," pattern-list ] ")".
list-pat         = "[" ([ pattern-list [ "," ".." ( ident | "?" )] ]
                    | ".." ( ident | "?" )) "]".
rec-pat          = "(." [ field-pat-list ] ".)".
field-pat        = ident ":" pattern.
var-id           = ident.
tp-id            = qualId.

def-pattern      = var-id ":" s-pattern
                 | var-id ":" tp-id
                 | var-id
                 | def-s-pat.
def-s-pat        = def-tup-pat | list-pat | rec-pat | literal | "?".
def-tup-pat      = "(" [ pattern-list ] ")".
```

A pattern describes the structure of a value and may contain variables. Variables match all values, whereas structured patterns match only values of the same structure. Type annotations may restrict the domain of matching values to the specified type. Nonlinear patterns are permitted, all values corresponding to the same variable must be equal. If a pattern matches a value, the variables of the pattern are bound to their corresponding values. Otherwise the match fails and no bindings are established.

Valid patterns are defined according to the paragraphs below. The following restrictions must be considered: Values of named types match only patterns that specify that type or an extended type. This guarantees name equivalence for named types. Each pattern must at least match one value. Each variable may at last be once aliased in a so called as-pattern of the form *id:s-pattern*. *id* must not occur in *s-pattern*. Each occurrence of the same variable must be untagged or tagged by the same type identifier. Type tags implied by declarations or named types need not be repeated in patterns.

6.1. Identifier Pattern

Each *var-id* matches each value; *var-id* is bound to the value.

An as-pattern *var-id: s-pattern* matches each value that matches the structured pattern *s-pattern*. *var-id* is bound to value.

6.2. Type Pattern

A type pattern *tp-id var-id* or *var-id:tp-id* matches each value of type *tp-id* or an extension of that type. *var-id* is bound to value.

A type pattern *tp-id s-pattern* matches each value of type *tp-id* or an extension that matches the structured pattern *s-pattern*. If *s-pattern* is a record pattern it must list only the field names of type *tp-id*.

A pattern *tp-id var-id:s-pattern* matches each value that matches *tp-id s-pattern*. *var-id* is bound to the value.

6.3. Tuple Pattern

A pattern *(pattern*$_1$*, ... , pattern*$_n$*)* matches each tuple with n components, if each component matches *pattern*$_i$.

6.4. List Pattern

A pattern *[pattern*$_1$*, ... , pattern*$_n$*]* matches each list with n components, if each component matches *pattern*$_i$.

A pattern *[pattern*$_1$*, ... , pattern*$_n$*, ..var-id]* matches each list with at least n components, if each component matches *pattern*$_i$. The rest of the list is bound to *var-id*. The type tag of a tagged list pattern applies to *var-id*.

6.5. Record Pattern

A pattern *(. ident*$_1$*: pattern*$_1$*, ... , ident*$_n$*: pattern*$_n$ *.)* matches each record with the specified fields, if the corresponding field values match. The record may have more fields than specified in the pattern.

6.6. Literal Pattern

Each literal and each constructor that consists only of literals is a valid pattern that matches only the designated value. "?" is a pattern that matches each value.

7. File Inclusion

module	= *"MODULE" ident ";" [import] [definition-sequ] "END" "."*.
definition	= *def | typeDef |declaration.*
import	= *"USE" ident-list ";"*.

The definitions of modules listed in a USE-list are included in the module and treated as if defined in the module.

8. Input and Output

A module WORLD offers a type WORLD and a variable w of that type. w represents the state of the file system. A module IO offers routines that allow for I/O operations. Each operation accepts a parameter of type world and gives a new value of the same type which represents the modified state of the I/O system. Each value representing an I/O state must be used only once. Hence, the state of the I/O system must be used linear.

Example:

```
FCT promptAndRead w:WORLD => (x, w3)
  WHERE VAL  w1:WORLD      => Write "Please enter a number: " w;
        VAL (x,w2:WORLD)   => Read w1;
        VAL  w3:WORLD      => Write "You entered: " w2;
        VAL  w4:WORLD      => Write x w3;
  END;
```

Given the following definition for SEQU, a more sophisticated implementation of promptAndRead is given below.

```
FCT  SEQU e [] => e;
FCT  SEQU e [f, ..fs] => SEQU (f e) fs;

FCT promptAndRead world:WORLD =>
```

```
SEQU (. x:0, w:world .)
  [ \p => (./p/ w: Write "Please enter a number: " p.w .),
    \p => (./p/ x:val,
                w:w' .)
                WHERE VAL (val,w') => ReadINT(p.w) END,
    \p => (./p/ w: Write "You entered: " p.w .),
    \p => (./p/ w: Write p.x p.w .)
  ];
```

A. Definition of Terms

belong to a type:

A value *w* belongs to type *T* if

1. *w* is T NIL.
2. *w* is numeric int constant (according to 1.2) and *T* is INT
3. *w* is numeric real constant (according to 1.2) and *T* is REAL
4. *w* is a string (according to 1.3) and *T* is STRING
5. *w* is TRUE or FALSE and *T* is BOOL
6. *T* is DYN
7. *T* is *named* and *w* belongs to *T* or an *extension* of *T.*
8. *T* is a structured *anonym* type, *w* structured value of the same kind with components belonging to the corresponding component types. For tuple types the number of components must be the same.
9. *T* is anonym record type, all fields of *T* are present in *w* and belong to the corresponding type.
10. *T* is anonym function type, *w* function of the same arity and all values of the declared result type of *w* belong to the result type of *T,* all values of the argument types of *T* belong to the argument types of *w.*

base type:

A record type R_1 is called base type of R_2 if R_2 is an *extension* of R_1.

range:

A range *a..b* consists of all numbers of type INT that are equal or greater than *a* and equal or less than *b*. If the order of the numbers is significant, the numbers are arranged in increasing order. If $a > b$ then the range is empty.

extension:

A record type R_1 is called direct extension of R_2 if there is a definition for R_1 like TYPE R_1 = (. (..., R_2, ...) ... .). R_1 is called extension of R_2 if R_1 is direct extension of R_2 or direct extension of an extension of R_2.

equal:

a and *b* are equal, if they belong to the same type and one of the following cases holds:

1. *a* and *b* are the same constants
2. *a* and *b* are tuples or lists with same number of components which are equal to the corresponding components.
3. *a* and *b* are sets, maps or records and all elements of *a* are members of *b* and vice versa. (Map and record associations are called elements in this definition)
4. *a* and *b* denote functions defined by the same function definition or lambda expression.
5. *a* and *b* are equal functions, applied to the same number of equal arguments.

smallest type:
: T is the smallest type a value w belongs to, if there is no type that is a subset of T that comprises w.

named type:
: A named type is introduced in a type definition.

specific:
: A pattern A is called more specific than B, if each value that matches A also matches B, and there is at least one value that matches B but not A.

overlap:
: Two patterns overlap, if there are values that match with both patterns, but no pattern is more specific than the other.

cast:
: A value w_a of type T_a can be casted into value w_b of type T_b if one of the following cases holds. T_a shall denote the *smallest* type that w_a *belongs* to.

1. T_a and T_b are the same type.
2. T_a is INT and T_b is REAL.
3. T_a and T_b are structured types of the same kind, T_a is anonym, and the components of w_a can be casted into the component types of T_b. Record fields not defined in T_b are ignored, i.e. w_a is projected to T_b. T_a may be a named record type, if T_a is an extension of T_b.
4. T_b is DYN.
5. w_a is NIL and T_b a recursive type.

B. Project Team

The following persons were involved in the implementation team and language design:

Dipl.-Inform. Michael Seyfried

Dipl.-Inform. Bernd Hofmann

Dirk Hinderks

Leif Kornstaedt

Thomas Oltzen

Claus George

Wolfgang Grund