Introduction to the Refal programming language

Alexander V. Konovalov
Bauman Moscow State Technical University
Moscow

ru-STEP: Russian seminar on Software Engineering,
Theory and Experimental Programming
Innopolis University, June 11 2021

History of the Refal programming language

- **1968**: first publication
  - Turchin V. F. Metaalgorithmic language. — Cybernetics #4, 1968, p. 116–124 (Турчин В. Ф. Метаалгоритмический язык. — Кибернетика № 4, 1968, с. 116–124)
- **1974** Basic Refal.
- **1986** Refal-2.
- **198x** Refal-5.
- **198x** Refal-6.
- **199x** Refal Plus
- **2016** Refal-5λ

History of metacomputations on Refal

- **1972–1974** two publications by Turchin:
  - Turchin V.F. Equivalent transformation of Recursive Functions defined in the language Refal, in: Trudy Vsesoyuznogo simposiuma «Teoriya Yazykov i Metody Programmirovania, Alushta,» Kiev, 1972, pp. 31–42
  - Turchin V.F. Equivalent transformation of REFAL programs, Avtomatizirovannaya Sistema Upravleniya Stroitelstvom. Trudy TsNIPIASS, GOSSTROY, Moscow, 1974, pp. 36–68
- **1980** Turchin V.F. The language Refal – The Theory of Compilation and Metasystem Analysis. Courant Computer Science Report, Num. 20 (February 1980), New York University
- **1981** Supercompiler SCP1 (Turchin V.F., Nirenberg R., Turchin D.V.)
- **1984** Supercompiler SCP2
- **1986** Turchin V.F. The concept of a supercompiler. ACM Transactions on Programming Languages and Systems. 8 (1986) 292–325, ACM Press
- **1987** Refal-4
  - Romanenko S.A. Refal-4 — Extension of Refal-2 Providing Expressiveness of the Results of the Driving, Preprint N147, IPM AN SSSR
  - Romanenko S.A. Driving for Refal-4 Programs, Preprint N211, IPM AN SSSR
- **1993** Supercompiler SCP3 (Turchin V.F., Nemytykh A.P.)
- **1999** Supercompiler SCP4 (Nemytykh A.P., supervised by Turchin V.F.)
- **2016** Supercompiler MSCP-A (Nepeivoda A.N., supervised by Nemytykh A.P.)

Short introduction to the Refal

- Name “Refal” means <u>Re</u>cursive <u>F</u>unctional <u>A</u>lgorithmic <u>L</u>anguage.
- Dynamically typed language.
- Base operation is pattern match such as Haskell or Erlang.
- The **program** is a sequence of functions.
- The **function** is a sequence of sentences (clauses).
- The **sentence** is a pair of pattern for argument and result expression.
- The **pattern expression** contains data constructors and variables.
- The **result expression** contains data constructors, variables and function calls.
- Thus, the syntax is similar to that of Haskell or Erlang.

Short introduction to the Refal

- Refal is dynamically typed language.
- The single data type is an object expression.
- The **object expression** is similar to LISP list: it is a sequence of terms.
- The **term** can be atomic value (“symbol”) or bracket term.
- The **bracket term** is object expression enclosed to parenthesis.
- The **symbol** can be character, word (such as LISP quoted name) or number. Some implementations can provide other symbol types.
- Examples of object expression
  - `'C' 'h' 'a' 'r' 's'`
  - `'Chars'              /* short equivalent of previous */`
  - `Two Words`
  - `((1 '+' 2) '*' (X '/' 3))`
  - `(Lisp McCarthy 1958) (Turchin Refal 1968) ("C++" Stroustrup 1980)`

Short introduction to the Refal

- **Variables** are written as `mode.Index`. Mode may be `s`, `t`, `e`, index is identifier or integer number. Variable modes:
  - s-variable can only be matched with one symbol,
  - t-variable can only be matched with one term (symbol or expression enclosed to brackets),
  - e-variable can be matched with any sequence of terms, including empty ones.
- We can compare Refal variables with filename wildcards (**`*`**`.txt`, `2021-`**`??`**`-`**`??`**`.zip`):
  - s- and t-variables are equivalent to `?` sign (singular character/term),
  - e-variables are equivalent to `*` sign (any string part).
- Examples expression with variables:
  - `e.BaseName '.txt'`
  - `e.Begin (s.Lang s.Author 1968) e.End`
  - `(t.Left s.Op t.Right)`

Short introduction to the Refal

- **Function calls** are enclosed to angle brackets: `<FuncName arg>`.
- Functions can take only one argument.
- Function definition has the syntax:

```
FunctionName {
  pattern = result;
  …
  pattern = result;
}
```

Short introduction to Refal

- Examples of the functions:

```
Factorial {
  0 = 1;
  s.N = <Mul s.N <Factorial <Sub s.N 1>>>;
}

ReplaceAtoB {
  'A' e.X = 'B' <ReplaceAtoB e.X>;
  s.1 e.X = s.1 <ReplaceAtoB e.X>;
  /* empty */ = /* empty */;
}

Rev {
  t.First e.Middle t.Last = t.Last <Rev e.Middle> t.First;
  t.One = t.One;
  /* empty */ = /* empty */;
}
```

Refal and other languages

- Compare Refal with some other languages. Refal function:

```
Factorial {
  0 = 1;
  s.N = <Mul s.N <Factorial <Sub s.N 1>>>;
}
```

- Haskell function:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- Erlang function:

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N - 1).
```

- There are similar, don’t it?

Refal and other languages

- What are the main differences between Refal and other languages?

Refal and other languages

- What are the main differences between Refal and other languages?
- Firstly, e-variables.
  - e-variables allow selects both the first and the last terms
    - `e.Begin t.Last`
    - `t.First e.End`
  - Pattern with e-variables may have ambiguous matching. It is important expressive feature.

Refal and other languages

- What are the main differences between Refal and other languages?
- Firstly, e-variables.
  - e-variables allow selects both the first and the last terms
    - `e.Begin t.Last`
    - `t.First e.End`
  - Pattern with e-variables may have ambiguous matching. It is important expressive feature.
- Secondly, repeated variables. If pattern has several variables with same names, they must have equal values.

Refal and other languages

- What are the main differences between Refal and other languages?
- Firstly, e-variables.
  - e-variables allow selects both the first and the last terms
    - `e.Begin t.Last`
    - `t.First e.End`
  - Pattern with e-variables may have ambiguous matching. It is important expressive feature.
- Secondly, repeated variables. If pattern has several variables with same names, they must have equal values.
- If we remove e-variables and repeated variables from Refal and require that the functions return one term, we get erlang-like language.

Refal and other languages

- Data in other languages are created from tuples with fixed arity. E.g. lists creates from cons-cells with arity 2.
  - Main operations are construction new tuple from k children and destruction tuple giving its elements.
- Refal data are trees with arbitrary count of children.
  - Main operations are concatenation, access to first and last children, or cuts it, iterate by children.

Refal and other languages

- Effective implementation of Refal data is a challenge for language implementor.
- Several approaches of implementation are known:
  - Flat double-linked lists (Refal-2, Refal-5, Refal-5λ)
  - Double-linked lists with hanging brackets (Refal-6, FLAC)
  - Arrays (Refal Plus)
- Each implementation have efficient (O(1)) and unefficient (O(|val|)) base operations (concatenation, create copy of value, etc)
- Perspective representations:
  - Ropes,
  - Finger trees,
  - Okasaki’s pure functional deques with concatenations.

Expressiveness of the patterns

- Repeated variables can represent values with equal parts.
  - `s.1 s.2 s.3` — pattern of three any symbols. Can be matched with `'abc'`, `1 2 3`, `True False False`, `'zzz'` etc.
  - `s.X s.X s.X` — pattern of three equal symbols. Can be matched with `'aaa'`, `7 7 7`, `True True True`.
- Equality comparison is a part of language core.

Expressiveness of the patterns

- Ambiguous patterns can perform complex queries.
- If pattern match is ambiguous, match result with shorten first e-variable is selected.
- If ambiguous match is not resolved, second, third… e-variables are checked.
- Example `'expressiveness' : e.1 's' e.2`. Match results:
  - ✔ `'expre' ← e.1, 'siveness' ← e.2`
  - ✕ `'expres' ← e.1, 'iveness' ← e.2`
  - ✕ `'expressivene' ← e.1, 's' ← e.2`
  - ✕ `'expressivenes' ← e.1, ε ← e.2`

Expressiveness of patterns

- Functions that replaces `'A'` to `'B'`:

```
ReplaceAtoB {
  'A' e.X = 'B' <ReplaceAtoB e.X>;
  s.1 e.X = s.1 <ReplaceAtoB e.X>;
  /* empty */ = /* empty */;
}
```

- can be rewritten shorter and to more effective:

```
ReplaceAtoB {
  e.X 'A' e.Y = e.X 'B' <ReplaceAtoB e.Y>;
  e.X = e.X;
}
```

Expressiveness of the patterns

- Set intersection by <u>one</u> recursive function:

```
/*
  <Intersect (e.Set1) (e.Set2)> == e.Intersect
*/
Intersect {
  (e.B1 t.Rep e.E1) (e.B2 t.Rep e.E2)
    = t.Rep <Intersect (e.B1 e.E1) (e.B2 e.E2)>;

  (e.Set1) (e.Set2) = /* empty */;
}
```

Expressiveness of the patterns

- More effective implementation:

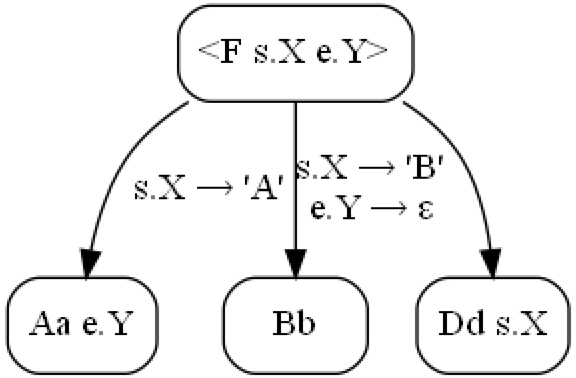
```
/*
  <Intersect (e.Set1) (e.Set2)> == e.Intersect
*/
Intersect {
  (e.B1 t.Rep e.E1) (e.B2 t.Rep e.E2)
    = t.Rep <Intersect (e.E1) (e.B2 e.E2)>;

  (e.Set1) (e.Set2) = /* empty */;
}
```

Refal and metacomputations

- **Metacomputations** are methods to analyze and transform programs by speculative execution.
- Two main tools of metacomputation are driving and generalization.
- **Driving** gives parametrized expression and perform one step of computations. The step may be ambiguous and different computation ways require different contractions and restrictions to parameters. Application a sequence of drivings to expression creates process tree.
- **Generalization** of two parametrized expressions is building new expression that original expressions are its special cases.

Refal and driving



- The expression

```
<F s.X e.Y>
```

- Source program

```
F {
   'A' e.B       = Aa e.B;
   'B'           = Bb;
   /* empty */   = Cc;
   s.A e.B       = Dd s.A;
}
```

Refal and driving

- Driving is performed by generalized pattern matching, special case of unification algorithm.
- For each sentence of driven function equation $\mathtt{E} \;:\; \mathtt{P}_\mathtt{n}$ are created, where $\mathtt{E}$ is argument of call, $\mathtt{P}_\mathtt{n}$ — n[th] pattern.
- Solution of equation is pair of substitutions: `Ct` than named "contractions" and `As` than named "assignments". `Ct` and `As` must satisfies the equation:

$$\mathtt{E} \;//\; \mathtt{Ct} \equiv \mathtt{P}_\mathtt{n} \;//\; \mathtt{As}$$

- Contractions label tree edges, assignments apply to right parts of sentences.

Refal and driving

- At 1972 Turchin formulated driving for Strict Refal. Strict Refal is Refal subset that repeated t- and e-variables are forbidden and pattern must be unambiguous.
- Driving of Strict Refal programs can be expressed in Strict Refal.
- Driving of Refal-5 with unrestricted patterns can’t be expressed in Refal-5.
- At 1987 Romanenko propose Refal-4 — extension of Refal-2 that driving transformation is closed on it. But driving algorithm was not proposed.
- Refal supercompilers SCP1, SCP2, SCP3 and SCP4 can transform programs in the Strict Refal (or subsets of Strict Refal).
- Model supercompiler MSCP-A (Nepeivoda, 2016—) is research of supercompilation with unrestricted patterns.

Refal and driving

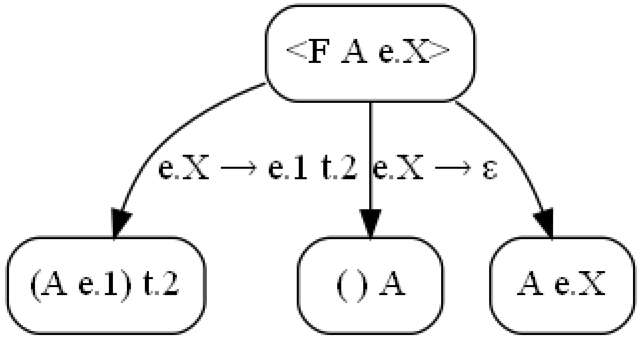
- Unlike other languages driving one sentence in Refal can provide several branches.

- The expression

```
<F A e.X>
```

- Source program

```
F {
  e.A t.B = (e.A) t.B;
  e.Z = e.Z;
}
```



Refal and driving

- Unlike other languages driving tree can have backtracks.

- The expression

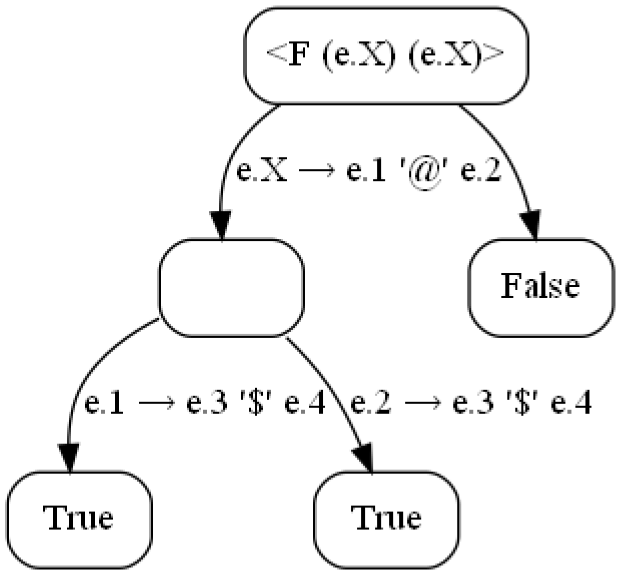
```
<F (e.X) (e.X)>
```

- Source program

```
F {
   (e.A '@' e.B) (e.C '$' e.D)
     = True;

   (e.AB) (e.CD) = False;
}
```

- Function `F` returns true if first subargument contains `'@'` and second one contains `'$'`.



Refal and driving

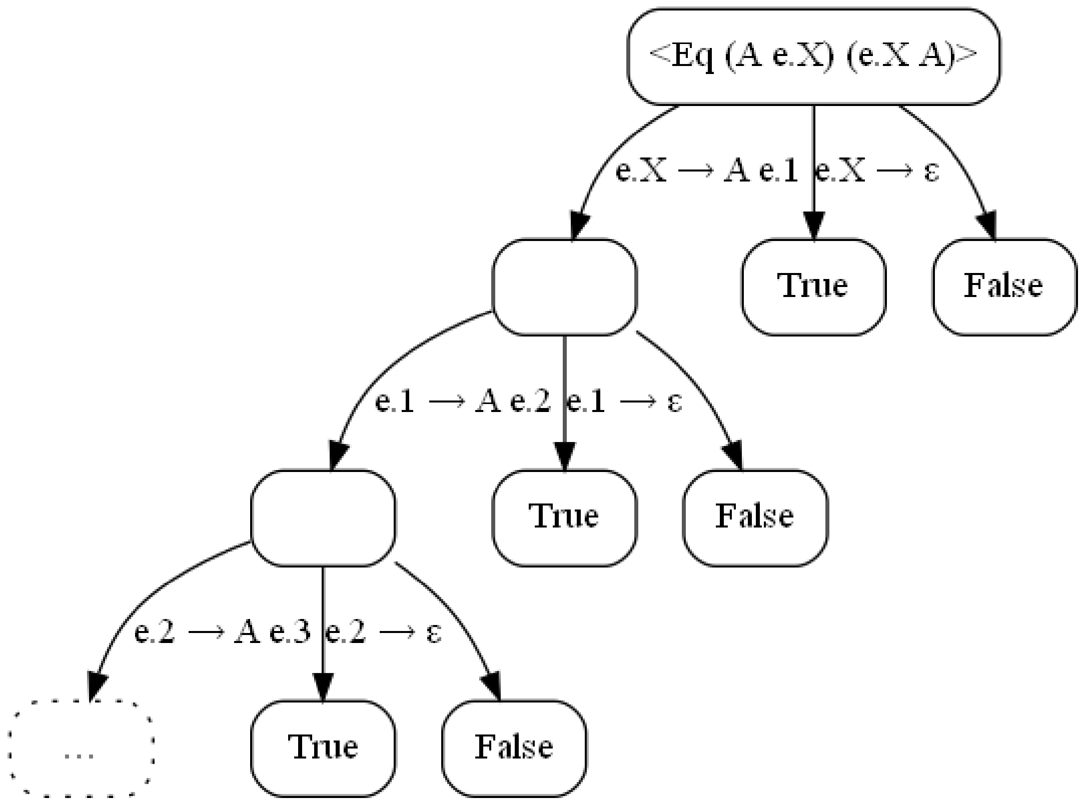
- Unlike other languages driving tree can be infinite.

- The expression

```
<Eq (A e.X) (e.X A)>
```

- Source program

```
Eq {
  t.X t.X = True;
  t.Y t.Z = False;
}
```



Refal and generalization

- Two parametrized expressions $\mathtt{E}_1$ and $\mathtt{E}_2$ are given. Generalization of it is expression $\mathtt{E}_G$ that there are substitutions $\mathtt{S}_1$ and $\mathtt{S}_2$ that

  $$\mathtt{E}_1 = \mathtt{E}_G \ // \ \mathtt{S}_1$$
  $$\mathtt{E}_2 = \mathtt{E}_G \ // \ \mathtt{S}_2$$

- **Most specific generalization** (MSG) is generalization $\mathtt{E}_G$ that no other generalization $\mathtt{E'}_G$ that there are non-trivial substitution $\mathtt{S}$ that

  $$\mathtt{E}_G = \mathtt{E'}_G \ // \ \mathtt{S}$$

- MSG is written as $\mathtt{EG = E1 \sqcap E2}$.

Refal and generalization

- Unlike other languages MSG in Refal is ambiguous:

```
A s.1 (e.2) ⊓ A (e.2) = A t.1 e.2
A s.1 (e.2) ⊓ A (e.2) = A e.1 (e.2)
A s.1 (e.2) ⊓ A (e.2) = e.1 s.2 (e.2)
```

- MSG in some cases may be unexpectable:

```
A e.1 ⊓ e.1 A = e.1 A e.2
```

Conclusion

- Patterns in Refal are more powerful that patterns in other languages.
- Effective implementation of Refal data is challenge for programmer.
- Base tools of metacomputations in Refal are not trivial.

Appendix

Implementation of union on Refal

Expressiveness of the patterns

- Set union by <u>one</u> recursive function:

```
/*
  <Union (e.Set1) (e.Set2)> == e.Union
*/
Union {
  (e.B1 t.Rep e.E1) (e.B2 t.Rep e.E2)
    = t.Rep <Union (e.B1 e.E1) (e.B2 e.E2)>;

  (e.Set1) (e.Set2) = e.Set1 e.Set2;
}
```

Expressiveness of the patterns

- More effective implementation:

```
/*
  <Union (e.Set1) (e.Set2)> == e.Union
*/
Union {
   (e.B1 t.Rep e.E1) (e.B2 t.Rep e.E2)
     = e.B1 t.Rep <Union (e.E1) (e.B2 e.E2)>;

   (e.Set1) (e.Set2) = e.Set1 e.Set2;
}
```