Introduction to Refal

Alexei Lisitsa

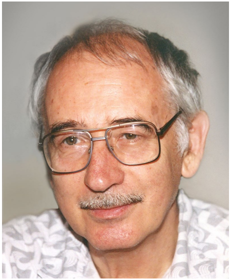
[1]Department of Computer Science
University of Liverpool,
Liverpool, UK

Meeting of the BCS Advanced Programming Specialist Group,
March 13, 2014

Outline

- Disclaimer
- Part I: Refal (Recursive Functions Algorithmic Language)
  - Valentin F. Turchin
  - Short Refal history outline
  - Metasystem Transitions and Metacomputation
  - Basic Refal
- Part II: Supercompilation
  - Main principles
  - Optimization
  - Verification
  - Normalization
- Conclusion

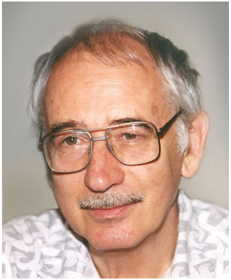
Valentin F. Turchin



1931-2010

- Computer Scientist and Physicist

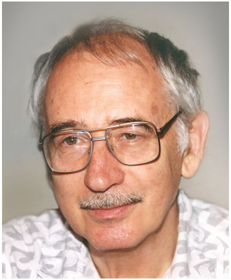
Valentin F. Turchin



1931-2010

- Computer Scientist and Physicist
- Philosopher

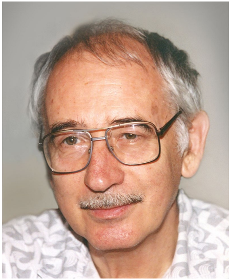
Valentin F. Turchin



1931-2010

- Computer Scientist and Physicist
- Philosopher
- Human Rights and Democracy Activist

Valentin F. Turchin



1931-2010

- Computer Scientist and Physicist
- Philosopher
- Human Rights and Democracy Activist
- Inventor of REFAL

Metasystem Transitions

V. Turchin:

- *Consider a system S of any kind.*

Metasystem Transitions

V. Turchin:

- *Consider a system S of any kind.*
- *Suppose that there is a way to make some number of copies of it, possibly with variations.*

Metasystem Transitions

V. Turchin:

- *Consider a system S of any kind.*
- *Suppose that there is a way to make some number of copies of it, possibly with variations.*
- *Suppose that these systems are united into a new system S' which has the systems of the S type as its subsystems, and*

Metasystem Transitions

V. Turchin:

- *Consider a system S of any kind.*
- *Suppose that there is a way to make some number of copies of it, possibly with variations.*
- *Suppose that these systems are united into a new system S' which has the systems of the S type as its subsystems, and*
- *includes also an additional mechanism which somehow examines, controls, modifies and reproduces the S-subsystems.*
- *Then we call S' a metasystem with respect to S, and the creation of S' a metasystem transition.*

Metasystem Transitions

V. Turchin:

- *Consider a system S of any kind.*
- *Suppose that there is a way to make some number of copies of it, possibly with variations.*
- *Suppose that these systems are united into a new system S' which has the systems of the S type as its subsystems, and*
- *includes also an additional mechanism which somehow examines, controls, modifies and reproduces the S-subsystems.*
- *Then we call S' a metasystem with respect to S, and the creation of S' a metasystem transition.*
- *As a result of consecutive metasystem transitions a multilevel hierarchy of control arises, which exhibits complicated forms of behavior."*

Metasystem Tansitions and Metacomputation

- V. Turchin, The Phenomenon of Science: a Cybernetic Approach to Evolution:
  - interpreted the major steps in biological and cultural evolution, as nothing else but metasystem transitions on a large scale.

V. Turchin:
Metacomputation is a computation which involves metasystem transitions (MST for short) from a computing machine M to a metamachine M' which controls, analyzes and imitates the work of M.

Refal

Refal was conceived as the universal language of metasystem hierarchies:

- on one hand, simple enough, so that the Refal machine could become an object of theoretical analysis;
- on the other hand, is rich enough to serve as a programming language for writing real-life algorithms.

First efficient interpreter for Refal: 1968 by V. Turchin.

Refal: short history

- In the 70-s the most popular implementation was Refal-2.
- During 80-s the language has been heavily revised and the three different implementations and dialects have been emerged:
  - Refal-5 (by V. and D. Turchin, New York),
  - Refal-6 (by N.Kondratiev and Ark.Klimov, Moscow)
  - Refal Plus (by S.Romanenko and R.Gurin, Moscow)

Refal: short history

- In the 70-s the most popular implementation was Refal-2.
- During 80-s the language has been heavily revised and the three different implementations and dialects have been emerged:
  - Refal-5 (by V. and D. Turchin, New York),
  - Refal-6 (by N.Kondratiev and Ark.Klimov, Moscow)
  - Refal Plus (by S.Romanenko and R.Gurin, Moscow)

For the puprose of this talk *basic Refal* will be used. It is a common subset (semantically) of all dialects above. We will use Refal-5 syntax.

Basic Refal Grammar

```
program::= $ENTRY definition+
definition::=function_name {sentence;+}
sentence::=left-side = expression
left-side::=pattern
expression::=empty|term expression|function-call expression
function-call::=<function-name arg>
arg::=expression
pattern::=empty|term pattern
term ::= SYMBOL|variable|(expression)
variable::=e.variable-name|s.variable-name|t.variable-name
empty::=/* nihil */

REFAL data are defined by the grammar
d::=d1 d2|(d1)|SYMBOL|empty
```

REFAL data

- `REFAL data are defined by the grammar`
  `d::=d1 d2|(d1)|SYMBOL|empty`
- REFAL data are terms
  - build from SYMBOLS, using
  - binary associative concatenation constructor (”blank”)
  - unary *no name* constructor (”brackets”)

REFAL data

- `REFAL data are defined by the grammar`
  `d::=d1 d2|(d1)|SYMBOL|empty`
- REFAL data are terms
  - build from SYMBOLS, using
  - binary associative concatenation constructor ("blank")
  - unary *no name* constructor ("brackets")
  - Examples:
    `(a (b A) B(B A)) ; (a A) (b B) (c C)`

REFAL data

- `REFAL data are defined by the grammar`
  `d::=d1 d2|(d1)|SYMBOL|empty`
- REFAL data are terms
  - build from SYMBOLS, using
  - binary associative concatenation constructor ("blank")
  - unary *no name* constructor ("brackets")
  - Examples:
    `(a (b A) B(B A)) ; (a A) (b B) (c C)`
  - Can be seen as *unranked* trees and hedges (forests)

REFAL profile

- Functional programming language;

REFAL profile

- Functional programming language;
- A strict programming language (the one in which only strict functions (functions whose parameters must be evaluated completely before they may be called) may be defined by the user);

REFAL profile

- Functional programming language;
- A strict programming language (the one in which only strict functions (functions whose parameters must be evaluated completely before they may be called) may be defined by the user);
- Operational semantics is based on pattern matching;

REFAL profile

- Functional programming language;
- A strict programming language (the one in which only strict functions (functions whose parameters must be evaluated completely before they may be called) may be defined by the user);
- Operational semantics is based on pattern matching;
- A function definition is a set of term rewriting rules, ordered from the top to the bottom;

REFAL profile

- Functional programming language;
- A strict programming language (the one in which only strict functions (functions whose parameters must be evaluated completely before they may be called) may be defined by the user);
- Operational semantics is based on pattern matching;
- A function definition is a set of term rewriting rules, ordered from the top to the bottom;
- When a function is called, a rule with a *topmost* matching condition is fired;

REFAL profile

- Functional programming language;
- A strict programming language (the one in which only strict functions (functions whose parameters must be evaluated completely before they may be called) may be defined by the user);
- Operational semantics is based on pattern matching;
- A function definition is a set of term rewriting rules, ordered from the top to the bottom;
- When a function is called, a rule with a *topmost* matching condition is fired;
- Variables are of one of *three* types:
  - e.Name (can take any expression as its value)
  - t.Name (can take any expression of the form (. . .) as its value)
  - s.Name (can take a symbol as its value)

Pattern Matching

- Pattern matching ≡ solving symbolic equations

Pattern Matching

- Pattern matching ≡ solving symbolic equations
- Associativity of concatenation ⇒ ambguity of pattern matching:

```
e.1 e.2 = A B

Possible solutions:
1) e.1 = [], e.2 = A B
2) e.1 = A, e.2 = B
3) e.1 = A B, e.2 = []
```

Refal's disambiguation rule:
Choose the solution with the minimal length of the datum assigned to the first e-variable (from left to right) and so on by induction.

Example of the program

```
$ENTRY Go {= <Prout<Pal 'revolver'>>}

           Pal  {           = True;
                         s.1 = True;
                         s.1 e.2 s.1 = <Pal e.2>;
                         e.1 = False;
                    }
```

- The prefix

  `$ENTRY`

  declares the function *Go* as an entry function;
- *Prout* is a built in function to print out the result of calling the function *Pal*
- What does *Pal* do?

Example of the program

```
$ENTRY Go {= <Prout<Pal 'revolver'>>}

            Pal  {           = True;
                          s.1 = True;
                          s.1 e.2 s.1 = <Pal e.2>;
                          e.1 = False;
                     }
```

- The prefix

  `$ENTRY`

  declares the function *Go* as an entry function;
- *Prout* is a built in function to print out the result of calling the function *Pal*
- What does *Pal* do? Checks whether the argument is a palindrome!

Second Example: Reverse

```
Reverse {
 = ;
 t.x  e.rest = <Reverse e.rest> t.x;
}
```

Third Example: Replacement

```
Fab {
a e.x = b <Fab e.x>;
s.1 e.x = s.1 <Fab e.x>;
(e.y) e.x = (<Fab e.y>) <Fab e.x>;
= ;
}
```

Markov Algorithms

- Refal execution model generalizes Markov normal Algorithms

Refal for XML processing

- Refal data are suspiciously similar to data in XML format

Refal for XML processing

- Refal data are suspiciously similar to data in XML format
- The same principle: *organize semantically significant substructures by using parentheses*

Refal for XML processing

- Refal data are suspiciously similar to data in XML format
- The same principle: *organize semantically significant substructures by using parentheses*
- Easy to convert XML data to Refal data: (roughly) replace `<tag ...>` with ((*tag*) and `</tag>` with)

Refal for XML processing

- Refal data are suspiciously similar to data in XML format
- The same principle: *organize semantically significant substructures by using parentheses*
- Easy to convert XML data to Refal data: (roughly) replace `<tag ...>` with ((*tag*) and `</tag>` with)
- Refal: the language for processing XML documents (V. Turchin)

Refal for XML processing

- Refal data are suspiciously similar to data in XML format
- The same principle: *organize semantically significant substructures by using parentheses*
- Easy to convert XML data to Refal data: (roughly) replace `<tag ...>` with ((*tag*) and `</tag>` with)
- Refal: the language for processing XML documents (V. Turchin)

Refal for XML processing

Query:

Collect the list of the tags in a given document for which a certain property named Property has the value True.

Refal program (V. Turchin):

```
Taglist {
((s.tag e.1 Property Is (True) e.2) e.x) = s.tag <Continue e.x>;
((s.tag e.properties) e.x) = <Continue e.x>;
}
Continue {
e.x (e.term) e.y = <Taglist (e.term)> <Continue e.y>;
e.x = ;
}
```

Supercompilation

- *Supervised Compilation*;
- Semantic based program transformation technique (V.Turchin, 1960-70s);
- Can be used for optimization and specialization of (functional) programs;
- Much of the development has been done in the context of Refal functional programming language;
- SCP4 is the most advanced implementation of supercompilation for Refal (Refal-5) (A.Nemytykh, V.Turchin).
- Supercompilers for other languages:
  -Java Supercompiler (A. V. Klimov),
  -Supero for Haskell (N. Mitchell)

Supercompiler

- observes the behaviour of a functional program P running on partially defined input;
- unfold a potentially infinite tree of all possible computations of P;
- reduce redundancy;
- folds the tree into a finite graph of parameterised configurations of P and transitions between them;
- based on a graph of configurations construct new program, which is (almost) equivalent to the input program.

Resulting program defines a function which is an extention of the input function.

Supercompilation for Optimization

Original program:

```
*$MST_FROM_ENTRY;
$ENTRY Go { e.ls = <pal e.ls <rev e.ls>>; }

pal {
     = True;

s.x = True;

s.x e.ls s.x = <pal e.ls>;
 s.x e.ls s.y = False;
}

rev {
          = ;
 t.x e.ls = <rev e.ls> t.x;
}
```

Supercompilation for Optimization (cont)

Suprcompiled program:

```
/*
$ENTRY Go {
 = <Prout <Go e.1 >> ;
}
*/

* InputFormat: <Go e.41 >
$ENTRY Go {
  = True ;
 s.102 e.41  = True ;
}

****************************** The End **********************
```

Verification via Supercompilation

- V.Turchin (1986):
  "... if we want to check that the output of a function $F(x)$ *always has the property* $P(x)$*, we can try to transform the function* $P(F(x))$ *into an identical* T *..."*
- The idea has not been tried until recently for the problems interesting for verification community.

Parameterised testing

General technique for verification of parameterised systems (A.Nemytykh, A.Lisitsa, 2005)

Parameterised testing

General technique for verification of parameterised systems (A.Nemytykh, A.Lisitsa, 2005)

- Let S be a parameterized system (a protocol) and P be a safety property of S to verify;

Parameterised testing

General technique for verification of parameterised systems (A.Nemytykh, A.Lisitsa, 2005)

- Let S be a parameterized system (a protocol) and P be a safety property of S to verify;
- Write a program φ_S simulating execution of S for n steps, where n is an input parameter;

Parameterised testing

General technique for verification of parameterised systems (A.Nemytykh, A.Lisitsa, 2005)

- Let S be a parameterized system (a protocol) and P be a safety property of S to verify;
- Write a program φ_S simulating execution of S for n steps, where n is an input parameter;
- If S is non-deterministic then let n be a string of characters labelling possible choices at the branching points;

Parameterised testing

General technique for verification of parameterised systems (A.Nemytykh, A.Lisitsa, 2005)

- Let S be a parameterized system (a protocol) and P be a safety property of S to verify;
- Write a program φ_S simulating execution of S for n steps, where n is an input parameter;
- If S is non-deterministic then let n be a string of characters labelling possible choices at the branching points;
- Given the value of n φ_s returns the state of the system S after execution n steps following the choices, provided by n;

Parameterised testing (cont.)

- Let T_P be a testing program, which given a state s of S returns the result of testing the property P on s (True or False);
- Consider composition $T_P \circ \varphi_S$. It first simulates the execution of the system and then tests the property required.

“P holds in any possible state reachable by the execution of the system S from an initial state”

$$\Leftrightarrow$$

"the program $T_P(\varphi(n))$ never returns the value *False*, no matter what values are given to the input parameter".

Practical implementation

- Refal is used to implement $T_P(\varphi(n))$
- SCP4 supercompiler is used to transform $T_P(\varphi(n))$ to a form from which one can easily establish required property by syntactical check:
    - resulting program does not contain the operator *return False;*

Refal examples (separate page)

MSI protocol

- Simplest cache coherence protocol: supports data consistency across multiple caches in shared memory multiprocessor systems
- Can be modelled by families of identical finite state machines with a primitive form of communication:
    - if one automaton makes a transition (an action) a, then it is required that *all* other automata make a complementary transition (reaction) $\bar{a}$
- The computation is assumed to be non-deterministic, i.e. at every step one automaton is choosen to make one of the available actions.
- Counting abstraction: keep track only of the number of automata in every possible (local) state.

Verification of parameterized protocols

- We would like verify the properties like

  If global machine for MSI of the dimension n starts in a global state with all automata in local states I then during any possible run

    - No two automata are simultaneously in the states S and M.
    - No two automata are simultaneously in the state M
- We would like to verify this property for all n.

Counting abstraction

- In protocols the automata are assumed identical ⇒ there is a lot of symmetry in their behaviour;
- Counting abstraction: keep track only of the *numbers of automata* in every possible (local) states.
  - For a broadcast protocol $\mathcal{P} = \langle Q, \Sigma, \bar{\Sigma}, \tau \rangle$, *configuration* of $\mathcal{P}$ is a function $c : Q \to N$;
  - Intuitively, $c(s)$ indicates how many processes are in the local state s
  - If $Q = \{s_1, \ldots, s_n\}$ then with any global state (of any dimension) one may associate configuration, presented as vector $(c(s_1, \ldots, c(s_n)) \in N^n$.

Extended FSM

Counting abstraction maps global machines for protocols into a variant of Extended FSM (Cheng, Krishnakumar 1997)

- States of EFSM are non-negative integer vectors;
- Transitions are guarded linear transformations;
- Guards are linear constraints.

EFSM for MSI

From the paper by E.A. Emerson and V. Kahlon (2003):

(PrWr1) $invalid \geq 1 \rightarrow invalid' = invalid + modified + shared - 1,\ modified' = 1,\ shared' = 0.$

(PrWr2) $shared \geq 1 \rightarrow invalid' = invalid + modified + shared - 1,\ modified' = 1,\ shared' = 0.$

(PrRd) $invalid \geq 1 \rightarrow invalid' = invalid - 1,\ modified' = 0,\ shared' = 1 + shared + modified.$

- The parameterized initial configuration is expressed as: $invalid \geq 1,\ modified = 0,\ shared = 0$
- The potentially unsafe states:
  - $invalid \geq 0,\ modified \geq 1,\ shared \geq 1$
  - $invalid \geq 0,\ modified \geq 2,\ shared \geq 0$

Refal encoding of EFSM for MSI

```
$ENTRY Go { e.time (e.i) =
 <Loop (e.time) (Invalid I e.i)(Modified )(Shared )>; }

Loop {
  () (Invalid e.1)(Modified e.2)(Shared e.3)
       = <Test (Invalid e.1)(Modified e.2)(Shared e.3)>;
  (s.t e.time) (Invalid e.1)(Modified e.2)(Shared e.3)
       = <Loop (e.time)  <RandomAction s.t  (Invalid e.1)
                                  (Modified e.2)(Shared e.3)>
          >;
}
RandomAction {
PrWr1 (Invalid I e.1) (Modified e.2) (Shared e.3)
                 = (Invalid e.1 e.2 e.3)  (Modified I)  (Shared );
PrWr2 (Invalid e.1)(Modified e.2)(Shared I e.3)
                 = (Invalid e.1 e.2 e.3)(Modified I)(Shared);
PrRd (Invalid I e.1)(Modified e.2)(Shared e.3)
                 = (Invalid e.1)(Modified )(Shared I e.2 e.3);
}
```

Test function

```
Test {
(Invalid e.1)(Modified I e.2)(Shared I e.3) = False;
(Invalid e.1)(Modified I I e.2)(Shared e.3) = False;
(Invalid e.1)(Modified e.2)(Shared e.3) = True;
}
```

- The potentially unsafe states:
  - $invalid \geq 0$, $modified \geq 1$, $shared \geq 1$
  - $invalid \geq 0$, $modified \geq 2$, $shared \geq 0$

Verification

- Now apply supercompiler SCP4 to the Refal program encoding the protocol composed with the testing function . . .

Verification

- Now apply supercompiler SCP4 to the Refal program encoding the protocol composed with the testing function . . .
- Residual program contains no *return False* operators;

Verification

- Now apply supercompiler SCP4 to the Refal program encoding the protocol composed with the testing function . . .
- Residual program contains no *return False* operators;
- Residual program never returns False;
- Original program never returns False;
- Specified protocol never violates the correctness condtion.

Results so far

Using the above scheme of Parameterised Testing + Supercomplication we have verified *parameterised* versions of:

- Cache Coherence Snooping Protocols (Synopsis N+1, MSI, MESI, MOESI, Illinois, Berkley, Dragon, ..);
- Cache Coherence Directory Protocols (Steve German's server-client protocol);
- Java MetaLock Algorithm;
- Load Balancing Algorirtm;
- Various Petri Nets models;
- . . .

Inductive theorem proving behind

LN 2007: a formal model which

- is simplified and refined theoretical version of SCP4;
- renders the supercompilation process in the specific context of verification tasks as an inductive proof;
- is formulated in terms of term rewriting systems.

Supercompilation vs Inductive Proving

- Refal program $\varphi_S \approx$ Term Rewriting System $\langle t, R \rangle$;
- Testing function $T_P \approx$ Property defined by Q_q;
- Residual program does not contain *return False* $\approx$ there is an inductive proof attempt with all vertices closed

Verification of various protocols

This approach has shown to be efficient for the verification of various (classes of) parameterised and infinite-state protocols and systems:

- Cache Coherence Snooping Protocols (Synopsis N+1, MSI, MESI, MOESI, Illinois, Berkley, Dragon, ...);
- Cache Coherence Directory Protocols (Steve German's server-client protocol);
- Java MetaLock Algorithm;
- Load Balancing Algorithm;
- Coverablity for Petri Nets;
- and others `http://refal.botik.ru/protocols/`

Formal Models and Results

- A simplified theoretical model of the verification via supercompilation approach
    - A.Nemytykh, A.Lisitsa, IJFCS Vol. 19, No. 04, 2008
- The completeness of the method for the verification of coverability for Petri Nets
    - A.Klimov, LNCS Vol. 7162, 2012,
    - A.Nemytykh, A.Lisitsa, RP'08, 2008 (announce)
- Verification using Java Supercompiler (A. Klimov)
- Verification of Cryptographic Protocols (A. Ahmed, A. Lisitsa, A. Nemytykh, 2008–2012)
- Normalization for metamorphic virus detection (A, Lisitsa, M. Webster, 2008)

Conclusion

- Refal is an interesting language:
    - Flexible data model

Conclusion

- Refal is an interesting language:
    - Flexible data model
    - Powerful pattern matching

Conclusion

- Refal is an interesting language:
  - Flexible data model
  - Powerful pattern matching
  - Supercompilation is available

Conclusion

- Refal is an interesting language:
    - Flexible data model
    - Powerful pattern matching
    - Supercompilation is available
    - Modern IDE and Eclips plugins are available (for Refal+ at least)

Conclusion

- Refal is an interesting language:
  - Flexible data model
  - Powerful pattern matching
  - Supercompilation is available
  - Modern IDE and Eclips plugins are available (for Refal+ at least)
  - Implementations are readily available: Refal-2, Refal-5, Refal-6, Refal+ ...

Thank you!