ISSN 0361-7688, Programming and Computer Software, 2020, Vol. 46, No. 1, pp. 57–66. © Pleiades Publishing, Ltd., 2020.
Russian Text © The Author(s), 2020, published in Programmirovanie, 2020, Vol. 46, No. 1.

Intermediate Representation of Programs with Type Specification Based on Pattern Matching

V. A. Vasenin[a],* **and M. A. Krivchikov**[a],**

[a]*Moscow State University, Moscow, 119991 Russia*

**e-mail: vasenin@msu.ru*

***e-mail: maxim.krivchikov@gmail.com*

Received May 13, 2019; revised June 8, 2019; accepted June 18, 2019

Abstract—In this paper, we present an intermediate representation (IR) language for the concise and generalized description of type system specification features in dynamically typed programming languages. The intermediate representation is based on pattern matching and features type-level computation. It is inspired by the intermediate representation "Assembly language" for Refal-2. In contrast to "Assembly language," the proposed representation is based on the control flow graph, which preserves typing information, rather than on bytecode. In addition, we propose a modification of the Refal language that supports higher-order functions, closures, and "associative array" data type, as well as a transpiler of programs from the language into the intermediate representation. In terms of this language, we present two examples of type system specification: simple types and row polymorphism. These examples are of interest for describing type systems for dynamically typed programming languages.

DOI: 10.1134/S0361768820010077

1. INTRODUCTION

Presently, domain-specific languages are generally implemented using dynamic type checking. As an example, we can point to some popular domain-specific languages. Shell languages of operating systems (Bash) and assembly systems (Make) use variable substitution in text format to describe arguments and results of programs they call. Template description languages, e.g., Jinja2, are implemented similarly. A number of domain-specific languages for describing configurations of software systems (Salt and Ansible) are based on markup languages (YAML and JSON) and do not support static type checking. For some domain-specific languages, e.g., the language for describing Nix software packages, the problem of replacing dynamic type checking with static one arises once they are developed and put into practice [1].

Static typing is supported by the following domain-specific languages: restricted versions of general-purpose languages (GLSL shader language as a dialect of C), languages built in general-purpose languages with more expressive type systems and type inference (various eDSLs in Haskell, e.g., those based on free monads), and languages implemented using macro systems of general-purpose languages (e.g., Rust macros).

The basic approach to implement type checking in dynamically typed programming languages is gradual typing [2]. For a more rigorous verification of properties in these languages, gradual typing systems often have certain features that are due to the well-established pragmatics (practice of use) of the language. For instance, the mypy tool, which implements gradual typing for Python, has the built-in support of the so-called structural typing protocols [3]. These protocols are specifications that the value must satisfy to be used in a particular syntactic context (e.g., as an iteration argument in the for...in statement or as an argument for square bracket indexing). The TypeScript language, which integrates gradual typing into JavaScript, supports the *keyof* keyword in type specifications. This keyword converts an object (associative array) into an enumeration of all known keys whose values are specified in this object.

In the framework of type theory, the most expressive approach to type description supports the calculus of constructions (CoC), which allows one to concisely and uniformly describe types by using type-level computations. In the CoC, types are defined, in fact, as terms of typed lambda calculus, i.e., as strictly terminating programs. In this paper, we propose an intermediate representation (IR) that uses type-level computations for the concise and generalized description of type system specification features in dynamically typed languages. More specifically, we use the intermediate representation based on pattern matching to describe type systems of programming languages. The results presented below were obtained in the work "Methods and tools for developing verifiable software by using domain-specific languages with a given formal semantics." The use of this representation for the description of domain-specific languages will be discussed in more detail in our future works.

2. LANGUAGE FOR DEVELOPING TOOLS TO DESCRIBE DOMAIN-SPECIFIC LANGUAGES

Currently, the most powerful (efficient) mechanisms for pattern matching in dynamically typed programming languages are implemented using languages of the Refal family [4] (in particular, Refal-5 [5]). Thus, as a basis for the intermediate presentation, we developed a modification of the Refal language with some extensions. This modification, which is described below, supports higher-order functions and "immutable associative array" data type. As far as we know, the latter feature is new to the languages of this family. The syntax of the language is modified in such a way as to make it more similar to currently-popular programming languages, which is also a new contribution. This section describes features of the modified language developed by the authors of this paper. The following sections describe the development and implementation of this language.

The basic type of constants in the language is lists, which can include the following elements separated by spaces.

(1) Non-negative integer constants (e.g., 0 12 12345). The current implementation supports 32-bit signed integers.

(2) String (multiline) constants in single or double quotes (as in Python, quotes are interchangeable) with the possibility to escape special symbols (e.g., "ABC" "A BC\" "'A\nB\tC'"). String constants are interpreted as sequences of individual characters (codepoints in Unicode terms).

(3) Symbols, i.e., identifiers that can be associated with a function (e.g., ABC and Fn1). From an implementation perspective, symbols can be regarded as interned strings.

(4) Nested lists separated by pairs of parentheses. For example, the list "A (B C) D" consists of three elements with the first element being symbol A, the second one being the list of two symbols B and C, and the third one being symbol D.

(5) Associative arrays written in curly brackets. The keys and values of the associative array can be single values. The key is separated from the value by a colon; the key–value pairs are separated by commas. For example, the associative array "{abc: DEF, ("12"): ("34"), 12: 34, (1 2): (3 4)}" contains the following four elements of the key–value type: abc symbol – DEF symbol, string (list of letters) "12" – string "34," number 12 – number 34, and list of numbers (1 2) – list of numbers (3 4). Other implementations of the Refal family languages do not support associative arrays and their pattern matching.

(6) Nested functions declared using the keyword *fn*, optional name, and body of the function in curly brackets.

Another class of expressions in the languages of the Refal family is patterns. Patterns are lists that, in addition to single values, can include free variables. Free variables are written as "type.name," as is customary in the other languages of the Refal family. The current implementation supports the following basic types of free variables:

• "s" is a single atomic value (number, symbol, or character);

• atomic value of a given type ("i" is a number, "S" is a symbol, and "c" is a character);

• "t" is a single arbitrary value (atomic value, list, or associative array);

• "d" is an associative array;

• "e" is an arbitrary sequence of values (sublist).

Patterns for nested lists are written in parentheses, while patterns for associative arrays are written in curly brackets. For pattern matching of associative arrays, the following additional syntactic constructs based on the ellipsis operator ("...") are introduced.

• Save the remainder of the associative array into a new variable. For instance, the construct {a: b, ...d.rest} defines a pattern for the associative array in which the value "b" should be under the key "a" and the other key–value pairs included in the array are stored in a new associative array, the variable d.rest.

• Ignore the remainder of the associative array. For instance, the associative array {a: b, c: d} does not match with the pattern {a: b} as it contains an extra key–value pair; however, it matches with the pattern {a: b, ...}.

For convenient reading of programs in the language, an empty list is denoted by the keyword "*empty*" both in values and in patterns. In addition, the dash ("_") is used to denote an unnamed arbitrary sequence of values.

The basic element of programs in the Refal family languages is the function. In the proposed implementation, function declaration begins with the keyword *fn*, followed by the name of the function and, then, a sequence of sentences (in curly brackets), which are pattern matching branches separated by semicolons (";"). The sentence begins with the pattern with which the function argument is matched; then, a combination of expressions and binding operators (see below) follows. The last expression in the combination determines the execution result of the sentence. In addition to constants and variables, expressions can contain calls of other functions. As in the other languages of the Refal family, the function call is enclosed in angle (activation) brackets. The first element in the brackets denotes the callee function, e.g., the call of the *Add* function for two numbers 1 and 2 is written as <Add 1 2>.

```
fn Palindrome {
  empty => True;
  s.1 => True;
  s.1 e.2 s.1 => <Palindrome e.2>;
  _ => False;
}
```

Fig. 1. Palindrome function in the proposed language.

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

Fig. 2. Palindrome function in Refal-5.

The classic example of the Palindrome function can be represented as follows. Figure 1 shows the code in the proposed language, while Fig. 2 depicts the corresponding example from the Refal-5 manual [5].

This example uses the binding operator "map" ("⇒"), which separates the expression to the left from the result (the expression to the right). The language supports the following binding operators.

• Operators of unconditional jump to the next step: comma (",") or map ("⇒"). In the other languages of the Refal family, as a binding element, the equality sign is generally used. If, when evaluating the expressions following this operator, an unsuccessful pattern match or unsuccessful function call occurs, then the execution of the function fails.

• The operator of conditional jump to the next step: ampersand ("&"). If, when evaluating the expression immediately following this operator, an unsuccessful pattern match or unsuccessful function call occurs, then the next pattern matching alternative within the active function is considered.

• The operator of simple additional pattern matching: "*let*" (*let* pattern = expression). It evaluates the expression and matches it with the pattern.

• The operator of additional pattern–function matching: "*match*" (*match* expression *with* function). It evaluates the expression and calls the function with the argument that is the value of this expression.

• A new negation operator "*except*" (*except* {sentence}). If the sentence in the initial context of variable values is evaluated successfully, then the negation operator returns a failure. Otherwise, the negation operator returns an empty list.

In general, the operators of additional pattern matching are similar to the conditions and blocks in Refal-5, while the split of jumps to the next step into conditional and unconditional ones is similar to that in Refal-6 or Refal-plus. The syntactic differences between this implementation and traditional ones are due to the convenience of using the language, in particular, the introduction to the language for the developers who are not familiar with the languages of the Refal family.

The restricted propagation of the information about an unsuccessful termination of computations within one function allows predicates to be defined on values as functions the evaluation of which is successful for the values that satisfy the predicate and is unsuccessful for the values that do not. In the following example, this feature is illustrated with the *PreAlph* function, which checks that the first argument (character) precedes the second one in a given alphabet. An implementation in Refal-5 returns symbol values *T* and *F* (true and false, respectively). Symbols *T* and *F* are not special ones (unlike, e.g., the *True* and *False* values in Python), and their use is entirely due to the agreement on the designation of Boolean value. An implementation in the proposed language regards the successful termination of computations (possibly with an empty result) as a true value and regards their unsuccessful termination (which can be intercepted using the binding operator "&") as a false value.

Thus, in the proposed implementation, two examples from Chapter 4 of the Refal-5 manual [5] can be written as follows, taking into account the syntactic and semantic features mentioned above (Figs. 3 and 4 show the code in the proposed language and Refal-5, respectively).

The negation operator is necessary for the adequate description of nontrivial conditions in type systems, which enables the intended use of the language. The example in Fig. 5 illustrates the use of the negation operator to determine the predicate "sequence of five atomic values, the third of which is not 1."

It should also be noted that the function declared in the match operator can use variables from the context of the external function's sentence in which the match operator was declared. In addition, this function can be declared with a name that can be used in it for a recursive call or return of a closure. Thus, the proposed implementation supports higher-order functions and closures. The example in Fig. 6 illustrates the language features mentioned above. The *Map* function is a higher-order function that applies the first function argument to each of the following arguments. The *AddOne* function defines a variable *s.add* that stores a certain value and an anonymous function stored in a variable *t.addOne*, which adds the *s.add* value from the closure to its own argument. This anonymous function is passed as the first argument to the *Map* function. Its execution yields a list the elements of which are incremented by one with respect to the original list (the list 1 2 3 4 5 becomes the list 2 3 4 5 6).

```
fn PreAlph {
   s.1 s.1;
   s.1 s.2 & let e.A s.1 e.B s.2 e.C = <Alphabet>;
}

fn Order {
   (e.1)e.2 & <Pre(e.1)(e.2)> => (e.1)(e.2);
   (e.1)e.2 => (e.2)(e.1)
}
```

Fig. 3. Predicates based on the successful/unsuccessful termination of computations in the proposed language.

3. INTERMEDIATE REPRESENTATION FOR THE DESCRIPTION OF DOMAIN-SPECIFIC LANGUAGES

Generally, to analyze programs, instead of their source code, some intermediate representation with a higher level of detail as compared to the source code (uniqueness of identifiers and expansion of syntactic sugar) and a higher level of hardware abstraction as compared to machine code is used. In [6], the following classification of intermediate representations for the description of domain-specific languages was proposed.

(1) Low-level intermediate representations used in compilers to machine code, e.g., LLVM representations and Typed Assembly Language.

(2) Bytecode-based application virtual machines designed for imperative programming languages, e.g., the JVM for Java and the CLI specification for C#.

(3) Bytecode-based application virtual machines designed for functional programming languages, e.g., the ZINC machine for OCaml, WAM machine for Prolog, BEAM machine for Erlang, and bytecode for Python.

(4) CFG-based internal intermediate representations that preserve typing information, e.g., continuation-passing style (CPS).

(5) High-level intermediate representations based on canonical forms of the syntax tree that reflect the specifics of a programming language, e.g., STG for Haskell and Cminor for CompCert (C compiler).

Based on this classification, we developed a new class 4 intermediate representation with a sufficiently expressive type system. It is inspired by the intermediate representation “Assembly language” for Refal-2.

The main differences between this intermediate representation and the assembly language can be formulated as follows. The assembly language belongs to the third class of intermediate representations, i.e., to bytecode-based application virtual machines focused on functional programming languages. The programs in the assembly language are sequences of instructions that do not have a deep internal structure. The proposed intermediate representation belongs to the fourth class, i.e., to CFG-based internal intermediate representations that preserve typing information. The assembly language is described for Refal-2; the proposed intermediate representation supports higher-order functions. As noted above, the language itself and, therefore, its intermediate representation are extended with new types and constructs for type system description. As a sufficiently close analog, we can point to the Refal-graph language, which is used in the SCP-4 supercompiler [8].

The execution model of the intermediate representation is as follows. When executing a function, at each instant, two lists of values (argument and result), a sequence of recognized patterns (variable values), and a set of variable values from the closure that are indexed by a pair of integers (the serial number for the external context of the closure and the number of the variable in the external context) are defined. The execution context of the function also includes a stack of

```
PreAlph {
   s.1 s.1 = T;
   s.1 s.2, <Alphabet>: e.A s.1 e.B s.2 e.C
         = T;
   e.1 = F;
}

Order {
   (e.1)e.2, <Pre (e.1)(e.2)>:
        { T = (e.1)(e.2):
          F = (e.2)(e.1);
        };
}
```

Fig. 4. Predicates based on return values in Refal-5.

```
ThirdOfFiveNot1 {
   s.1 s.2 s.3 s.4 s.5,except {let 1 = s.3}
}
```

Fig. 5. Negation operator: a predicate for a sequence of five atomic values, the third of which is not 1.

handlers for unsuccessful recognitions or function calls. The result can contain call instructions (activation brackets). Before saving it into a variable or before analyzing the pattern, activation is performed. At the activation stage, all call instructions in the list are replaced by call results.

The intermediate representation is stored as a tree in the text form of s-expressions; it should be noted that, if necessary, the storage format can easily be changed, e.g., to compact non-textual code or XML. The tree contains the expression nodes of the following main types.

(1) Function declaration (*Function* (*Name* name-of-function) expressions...) and reference to the function (*FunctionRef* function name).

(2) List or structural brackets (*Structural* expressions...).

(3) Call or activation brackets (*Eval* expressions...).

(4) Negation (*Except* expressions...).

(5) Branching (*Branch* (expressions of branch 1) (expressions of branch 2)...).

(6) Associative array as a value (*Dictionary* ((expression-key) (expression-value))...).

(7) Constant as a value (*Value* value-type value).

(8) Instructions without arguments: *Check* (recognizing an empty argument), *Return* (successful execution of the function), *Fail* (unsuccessful termination of the computation branch), *PopHandlers* (clearing the handler stack), *Expand* (finding an arbitrary sequence of values), *Close* (recognizing the remainder), and *Push* (activating and saving the current result in a sequence of variables).

(9) Instructions for recognizing a fragment of an argument (*Recognize* direction recognizer). The following directions are supported: the beginning and end of the argument (*Left* and *Right*, respectively), inclusion in the associative array at the beginning of the argument (*Dictionary*), as well as key and value in the associative array (*Key* and (*Value* number of variable-key), respectively). Special directions (*Dictionary* and *Key*) are supported only for some recognizers. As recognizers, nested lists (*Structural*), arbitrary values (*Term*), atomic values (*Literal*), values of a given primitive type (*ValueLiteral*), constants (*Exact*), and values of previously recognized variables (*OldExpression* level number) are supported.

(10) Instruction for finding an arbitrary sequence of values (*Extend*).

(11) Instruction for loading the variable value into the argument to impose additional conditions (*Range* variable-number).

(12) Instruction for yielding constant values (*Value* type value) and values of previously recognized variables (*Emit* level number).

(13) Type check instructions (*Typecheck IsType* function-name) and (*Typecheck Correct* type-precondition function-name type-postcondition).

The field "level" in the *OldExpression* recognizer and *Emit* instruction is zero when the number corresponds to the variable in the context of the current function. Otherwise, for closures, it reflects the nesting level of the lexical context from which the variable with this number is loaded.

This intermediate representation is sufficient for the translation of programs written in the language described in the previous section. As one of the directions for further research, we intend to achieve higher uniformity of the IR instructions. As noted above, the recognition directions *Dictionary*, *Key*, and *Value* differ significantly in terms of semantics from the directions *Left* and *Right*. In particular, they cannot efficiently represent the enumeration of key—value pairs included in the associative array to recognize associative arrays with complex structures. Currently, to solve this problem, we use relatively less efficient functions of the standard library that perform the conversion between the associative array and the list of key—value pairs constituting the array.

In addition, with the intermediate representation being designed to describe the semantics of programs written in domain-specific languages, the IR instructions can contain debugging information. This information relates source code elements and IR instructions. Currently, the format of this additional information is not yet strictly defined, which is also one of the directions for further research.

4. IR INTERPRETER

For the execution of the programs written in the intermediate representation, we developed an IR interpreter. The basic requirements for the interpreter were its extensibility and performance sufficient for the adequate support of the development cycle in research mode. Thus, we decided to implement the language executor in the form of an interpreter rather than a compiler to machine code. The interpreter is implemented in Rust with its size being about five thousand lines of code.

The interpreter uses an extended (executable) version of the intermediate representation for more efficient interpretation. The stored version of the intermediate representation is converted into the executable one at the preprocessing stage. At that stage, the interpreter reads the IR tree from the source file and saves it as an arena, a set of linear instructions sequences indexed by a 32-bit unsigned integer. Thus, in memory, the instructions such as *Structural* or *Function* store the index of the instruction sequence in the arena. The main purpose of the preprocessing stage is to carry out the following two procedures. First, the names of functions in *FunctionRef* instructions are associated with the indexes of their definitions in accordance with the lexical context. Second, for closures, the numbers of the variables used by the closure are extracted, and the *Function* instruction is replaced by an interpreter-specific instruction for capturing variables in the closure (*CaptureClosure*). In the future, we intend to move this instruction to the stored part of the intermediate representation. Currently, however, the transformation mentioned above is technically easier to implement as part of preprocessing in the interpreter rather than in the translator to the intermediate representation. In addition, the preprocessing procedure loads external modules and binds imported functions. This aspect of the implementation is technical in nature and its description falls outside the scope of this paper.

The following two features of the proposed language, which are used in the intermediate representation and have a significant effect on the structure of the interpreter, should also be noted. The first feature is associated with the execution of individual functions. In many currently-popular programming languages, the semantics of the *Branch* instruction can be adequately described by the conditional statement "if." More specifically, if some condition is not satisfied, then a jump to the next branch of the conditional statement occurs. However, the *Extend* instruction has nontrivial semantics, which is more common to logic programming languages and is due to the semantics of *e*-type variables in the languages of the Refal family. All cases of unsuccessful pattern matching or function calls that occur after the execution of the *Extend* instruction return control to the *Extend* handler, which rolls back the computations. The handler restores the original value of the argument and appends the next element of the argument to a variable; then, it resumes the execution of the operations following the *Extend* instruction with the new value of the argument. To adequately describe this behavior in terms of structural programming, we can use the *while* statement. Finally, the semantics of the *PopHandlers* instruction, which corresponds to the binding operator of unconditional jump, together with the *Extend* and *Branch* instructions described above, can be adequately described only in terms of direct jumps to command blocks.

The second feature is associated with the semantics of processing unsuccessful function calls. In the classic implementations of Refal-2 and Refal-5, an unsuccessful function call causes the program to crash. The proposed implementation uses a principle similar to software exception handling in C++, C#, and Java. More specifically, the unsuccessful function call returns an unsuccessful pattern matching signal at the level of the caller function. Hence, this signal can be processed in a standard way using the *Branch* and *Extend* handlers. Such semantics can be compared with rollback functions in Refal-plus [9]. The difference is that the decision about processing or not processing the unsuccessful function call is made at the level of the caller function rather than at the level of definition. Thus, the semantics of this aspect of the language is more similar to exception handling rather than to rollback functions. Depending on the instruction sequence of the caller function, the result can be activated both in the context of the caller function and one level higher in the call stack. The latter takes place for the return value of the function if it is specified after the unconditional jump operator. The current implementation of the IR interpreter supports this feature in the form of partial tail recursion.

The memory management features of the interpreter should also be noted. The interpreter of the intermediate representation described in this paper uses automatic memory management. Lists of values are stored based on reference counting. To improve efficiency, the list of values can be stored both as a continuous sequence of values (or a reference to a fragment of another list) and as a sequence of fragments. When adding values of recognized variables to the list by using the *Emit* instruction, the following heuristic is employed.

(1) If the old and new values are both successive fragments of the same list, then, instead of the pair of these fragments, one fragment containing all necessary values is written.

(2) Otherwise, if the length of a newly added fragment exceeds the threshold, then the fragment is added as a reference. The threshold is determined experimentally for more efficient use of cache memory in modern processors.

(3) If the length of a newly added fragment does not exceed the threshold, then its elements are copied to a new list.

At the early stages of developing the interpreter, the mark-and-sweep garbage collection algorithm was employed for automatic memory management, both for values and the IR code. However, its overhead proved too high. Thus, the garbage collection algorithm was modified to use a two-generation scheme. This modification provided a several-fold speedup; however, the performance of the interpreter was still insufficient with unreasonably high memory consumption. Eventually, replacing garbage collection with a static arena for code storage and counting references for value fragments yielded a two-order speedup (as compared to the mark-and-sweep algorithm with two generations) and sufficient (for now) performance.

In its current version, the interpreter translates the translator (see the next section) from the source language into the intermediate representation in 1.1 seconds while using 9.4 MB of memory. The latest version of the translator, which can be executed by an external Refal implementation used as a starting point for developing the self-hosting implementation of the language, is compilable into C. It performs the corre-

```
fn Map {
  t.fn => emply;
  t.fn s.1 e.rest => <t.fn s.1> <Map t.fn e.rest>;
}

fn AddOne {
  _ =>
    let s.add = 1,
    let t.addOne = fn{s.val => <Add s.add s.val>},
    <Map t.addOne 1 2 3 4 5>
}
```

Fig. 6. Higher-order function and closure in the proposed language.

sponding translation in 1.57 seconds while using about 1 GB of memory.

With time, the efficiency of cache memory usage (and, therefore, the performance of the interpreter) can be further improved based on a more concise representation of value and instruction sequences in memory. Currently, however, we believe that the performance of the interpreter is sufficient and consider it unreasonable to complicate the code of the interpreter to boost its performance.

Below, we mention some previous versions of the interpreter that were developed for experimental purposes. A simple recursive interpreter of instruction trees proved inadequate due to the lack of support for guaranteed tail recursion in popular programming languages. As a result, for nontrivial programs, there was always a possibility of stack overflow. A translator into JavaScript code was developed on a consideration that modern language implementations based on tracing JIT compilers (we used a V8 implementation) provide sufficient performance for a relatively simple translation scheme. In practice, however, this implementation proved invalid due to some specific features of the garbage collection algorithm in the V8 machine and high memory consumption. An interpreter in Haskell was developed as an executable model for the denotational semantics of the language. However, the performance sufficient for our research and a comparable volume of implementation code in Rust suggested that there is no urgent need to refine this version.

5. DEVELOPING A TRANSLATOR FROM THE SOURCE LANGUAGE TO THE INTERMEDIATE REPRESENTATION

The intermediate presentation described in this paper is designed, like any other intermediate presentation, to be used by automated tools, rather than to write programs manually. To test the interpreter and implement the Refal modification described above, we developed a translator from this language to the intermediate representation.

The structure of the translator is similar to the standard one. The source code passes the stages (transformations) described below.

1. Lexical analysis: a string of characters is divided into uniform sequences called tokens (constants, identifiers, punctuation marks, etc.).

2. Bracket extraction: in the sequence of tokens, the structure based on nested pairs of brackets corresponding to one another is extracted.

3. Parsing: the bracket-structured tree of tokens is transformed into an abstract syntax tree of the source language.

4. Translation to the intermediate representation: the abstract syntax tree is transformed into a sequence of IR instructions. The main purpose of the translation is to convert patterns into a sequence of instructions, which is similar to the algorithm of translation to the Refal assembly language [7] with some simplifications. With the memory management method used (counting references to fragments), which allows new copies of list fragments to be inserted in constant time, there is no need to use nontrivial transformations to optimize the output at this stage.

5. Convolution of recognition branches: sequences of instructions that are common prefixes for all recognition branches in the code of a function are taken out of the branch instruction.

6. Local optimization: a relatively simple IR translation algorithm generates code in which some patterns that allow for further simplification can be extracted. For instance, in the sequence of instructions *(Close) (Check)*, the latter instruction always terminates successfully, which is why it can be excluded while preserving semantics. At the local optimization stage, such patterns in the command tree are reduced to a simpler form.

7. IR serialization to the output text format: at this stage, line escaping is carried out and the intermediate representation is displayed as text with the indentations that reflect the structure of the code.

The first version of the translator was developed in Refal-5 and ran on one of its implementations freely

```
fn Bool {
   0;
   1;
}

fn Not {
   0 => 1;
   1 => 0;
}

fn IncompleteNot {
   0 => 1;
}
```

Fig. 7. Code of functions over Boolean variables.

available on the Internet. Then, the translator was put (using the bootstrapping method) in self-hosting mode based on the following scheme.

Step 1. The translator T_0 written in Refal-5 is compiled by the Refal-5 compiler to the executable code E.

Step 2. Iterative refinement of the interpreter and translator (T).

a. The intermediate representation and machine code of the translator's fragment (E(T') = I', Refal-5(T') = E') are obtained. As these fragments, the code of each translation stage is used sequentially.

b. The intermediate representation is passed as an input to the interpreter for self-application (Interpreter(I')(T')). The result of its execution is compared with the execution result of the machine code (E'(T')).

c. If any error or inconsistency occurs, then the interpreter or translator is refined.

d. The iteration terminates once self-hosting is achieved:

E(T) = I,

Interpreter(I, T) = I_1,

Interpreter(I_1, T) = I_2,

Interpreter(I_2, T) = I_3,

$I_2 \equiv I_3$

The iterative refinement is carried out manually due to the specifics of the problem. The self-hosting check by scenario d is added to the automatic test suite and is carried out with each modification of the interpreter or translator. Once self-hosting is achieved, the initial implementation of Refal-5 is no longer required to run the modified translator, which allows us to go to the next step.

Step 3. Iterative modification of syntax and semantics.

a. Transitional syntax support is added to the interpreter.

b. The translator's code T is translated to the transitional syntax.

c. Support for the old syntax is removed and support for the new syntax is added.

d. The translator's code T is translated to the new syntax.

At each stage, the preservation of the self-hosting state is checked by directly testing the condition specified in substep d of step 2.

6. IR-BASED DESCRIPTION OF SIMPLE TYPES AND ROW POLYMORPHISM

This section provides some examples of using the developed intermediate representation to describe simple types.

An example for simple types is shown in Figs. 7 and 8. The *Bool* function specifies a data type that can take one of two values: 0 or 1. If the *Not* function is called with an argument that satisfies the *Bool* specification, then the evaluation of this function is guaranteed to complete successfully with the evaluation result satisfying the *Bool* specification. In turn, the *IncompleteNot* function is not defined for some possible values of the argument that satisfies the *Bool* specification. For these functions, these results can be obtained automatically using a partial evaluation (driving) algorithm similar to that described in [10]. It should be noted that, currently, the driving algorithm is implemented only for some IR constructs. The implementation of the driving algorithm used in this work evaluates the function on a given initial parameterization of the argument and returns two sets of parameterizations. The first (positive) set contains all possible refinements of the initial parameterization on which the function is evaluated successfully, as well as the representation of the evaluation result in each refinement. The second (negative) set supplements the first one; i.e., it describes all possible refinements of the initial parameterization on which the evaluation process fails.

Row polymorphism is a property of the type system that allows functions to be defined on associative arrays for which the type system guarantees the impossibility of exceptions such as "key is absent in the associative array" and "value type does not match the expected one" [11].

To define row polymorphism, it is sufficient to introduce a row type and three operations—projection (*Select*; to obtain the value by the key), row addition (*Add*), and row removal (*Remove*)—as shown in Fig. 9. To describe the row type, the associative array is used. The behavior of the *Select* function is described using axiomatic semantics in terms of Hoare triples. The precondition *SelectPreCondition* and postcondition *SelectPostCondition* characterize the behavior of the *Select* function, namely, the fact that, for any (decidable) property *s.predicate*, if the condition *SelectPreCondition s.predicate* is satisfied for some set of values, then the *Select* function with this argument is evalu-

```
(Function (Name Bool) (Branch
  ((Recognize Left Exact Int 0)
    (Check)(Return))
  ((Recognize Left Exact Int 1)
    (Check)(Return))
))

(Function (Name Not) (Branch
  ((Recognize Left Exact Int 0)
    (Check)(Value Int 1)(Return))
  ((Recognize Left Exact Int 1)
    (Check)(Value Int 0)(Return))
))

(Function (Name IncompleteNot) (Branch
  ((Recognize Left Exact Int 1)
    (Check)(Value Int 1)(Return))
))
```

Fig. 8. Intermediate representation of functions over Boolean variables.

ated successfully and the condition *SelectPostCondition s.predicate* is satisfied for the result.

In terms of the driving algorithm, taking into account its current limitations, the process of checking this condition can be described as follows. Suppose that the evaluation of the function *s.predicate* always ends with a successful or unsuccessful result. Let us run a driving for the precondition with an arbitrary argument, <*SelectPreCondition s.predicate e.argument*>, and retain only the positive set of argument parameterizations. For all positive parameterizations *P* of *e.argument*, we run a driving for the original function: <*Select P*>. The driving should yield only positive parameterizations. Otherwise, the precondition sets too weak constrains under which the successful completion of the *Select* function cannot be guaranteed. For all positive parameterizations *Q* (results of the second driving), we perform a driving for the postcondition <*SelectPostCondition s.predicate Q*>. As with the second one, this driving should yield only positive parameterizations.

As a direction for further research, we plan to extend the driving algorithm to enable the full support of the intermediate representation. In addition, we intend to enrich the source language with two constructs for type checking. These constructs correspond to the type checking instructions in the intermediate representation; below is their preliminary versions.

1. The construct "*type* expression" is used to verify that the expression is a type, i.e., it runs a static terminability check of the expression for any input values.

2. The construct "*correct* (precondition) (function) (postcondition)" runs the check described above for the types "precondition" and "postcondition."

7. CONCLUSIONS

One of the most important practical aspects of the programming language is communication between

```
fn Select {
  s.label {s.label: t.value, ...} => t.value
}

fn SelectPreCondition  {
  s.predicate s.label { s.label: t.value, ...},
    <s.predicate t.value>
}

fn SelectPostCondition {
  s.predicate t.value, <s.predicate t.value>
}

fn Add {
  {...d.dict} t.value s.label,
    excepr{let {s.label: _,...}=t.dict}
    => {...d.dict,s.label: t.value}
}

fn Remove {
  s.label{s.label: _,...d.rest} => {...d.rest}
}
```

Fig. 9. Implementation of row polymorphism.

developers. The source code contains the most accurate and up-to-date information about the internal structure of the system. For new programming languages, the similarity of their syntax with other languages the developer is familiar with contributes to a faster understanding of the code written in this language [12]. In addition, according to [12], a language that does not support keywords or actively uses nonobvious special symbols (instead of keywords) seems less intuitive than a language with keywords. This is confirmed by our experience with students of the Faculty of Mechanics and Mathematics of the Moscow State University.

Among the analogs of the proposed type checking approach, we want to mention the work by V.I. Shelekhov on predicate programming [13]. The languages of the Refal family and supercompilation were previously used to verify properties that have the nature of low-level hardware abstraction [14, 15]. In [16], supercompilation was used in the framework of the Martin–Löf type theory. In the general context of the research the results of which are presented in this paper, we should mention the language of executable software specifications [17], which is designed for the specification of domain-specific languages. Function-based description of types was also used in the prototype verification system (PVS) [18], which was employed in the work mentioned above [13].

As a direction for further improvement of the syntax, we consider the possibility of using a more conventional notation for lists and pattern variables. In many popular programming languages, lists are denoted by square brackets. An exception is the languages of the LISP family, which can be considered relatively less popular. Using square brackets for lists allows parentheses to play their generally accepted role, i.e., denote grouping constructs. To denote variables, we intend to use only their names with optional type annotation in brackets. In this case, variables that denote recognition patterns of an arbitrary sequence are denoted by the ellipsis prefix, e.g., the pattern (s.a e.b s.a) is written as [a ... b a] or [(a: atom) ... b a]. The problem of the notation that distinguishes symbols and pattern variables requires further research. One of the possible solutions to this problem is the case-based representation of the identifier (as in Haskell): the first letter of the variable is lowercased, while that of the symbol is uppercased.

8. FUNDING

This work was supported by the Russian Foundation for Basic Research, project no. 16-07-01178a.

REFERENCES

1. Hufschmidt, T., A type-system for Nix. http://nixcon2017.org/schedule.nixcon2017.org/system/event_attachments/attachments/000/000/003/original/main.pdf.
2. Siek, J. and Taha, W., Gradual typing for objects, *Proc. European Conf. on Object-Oriented Programming (ECOOP),* Ernst, E., Ed., Berlin: Springer, 2007, pp. 2–27.
3. Levkivskyi, I., Lehtosalo, J., and Langa, L., PEP 544 – Protocols: Structural subtyping (static duck typing). http://www.python.org/dev/peps/pep-0544. Accessed January 10, 2019.
4. Turchin, V.F., Meta-algorithmic language, *Kibernetika,* 1968, no. 4, pp. 45–54.
5. Turchin, V.F., *REFAL-5: Programming Guide and Reference Manual,* Holyoke: New England, 1989.
6. Vasenin, V.A. and Krivchikov, M.A., Methods for intermediate representation of programs, *Program. Inzheneriya,* 2017, vol. 8, no. 8, pp. 345–353.
7. Romanenko, S.A., Machine-independent compiler from the recursive function language, *Cand. Sci. (Fiz.-Mat.) Dissertation,* Moscow: Inst. Prikl. Mat. Keldysha, 1978.
8. Nemytykh, A.P., *Superkompilyator SCP-4: obshchaya struktura* (SCP-4 Supercompiler: General Structure), Moscow: Izd-vo LKI, 2007.
9. Romanenko, S.A. and Gurin, R.F., *Yazyk programmirovaniya Refal Plyus* (Refal Plus Programming Language), Pereslavl'-Zalesskii: Univ. Pereslavlya im. A.K. Ailamazyana, 2006.
10. Turchin, V.F., *Ekvivalentnye preobrazovaniya rekursivnykh funktsii, opisannykh na yazyke REFAL* (Equivalent Transformations of Recursive Functions Described in the REFAL Language), Kiev-Alushta, 1972, pp. 31–42.
11. Wand, M., Type inference for record concatenation and multiple inheritance, *Inf. Comput.,* 1991, vol. 93, no. 1, pp. 1–15.
12. Stefik, A. and Siebert, S., An empirical investigation into programming language syntax, *Trans. Comput. Educ.,* 2013, vol. 13, no. 4, pp. 19:1–19:40.
13. Shelekhov, V.I., Verification and synthesis of efficient programs of standard functions in the predicate programming technology, *Program. Inzheneriya,* 2011, no. 2, pp. 14–21.
14. Lisitsa, A.P. and Nemytykh, A.P., Verification as a parameterized testing (experiments with the SCP4 supercompiler), *Program. Comput. Software,* 2007, vol. 33, no. 1, pp. 14–23.
15. Lisitsa, A.P. and Nemytykh, A.P., On one application of computations with oracle, *Program. Comput. Software,* 2010, vol. 36, no. 3, pp. 157–165.
16. Klyuchnikov, I.G. and Romanenko, S.A., Supercompilation for Martin-Lof's type theory, *Program. Comput. Software,* 2015, vol. 41, no. 3, pp. 170–182.
17. Novikov, F.A. and Novoseltsev, V.B., Interpretable program specification language, *Program. Comput. Software,* 2010, vol. 36, no. 1, pp. 48–57.
18. Shankar, N. and Owre, S., Principles and pragmatics of subtyping in PVS, *Recent Trends in Algebraic Development Techniques,* Bert, D., Choppy, C., and Mosses, P.D., Eds., Springer, 1999, vol. 1827, pp. 37–52.

Translated by Yu. Kornienko