Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Interner Bericht

Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL'03)

Jörg Striegnitz, Kei Davis (Eds.)*

FZJ-ZAM-IB-2003-10

FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL'03)

*Jörg Striegnitz, Kei Davis** *(Eds.)*

FZJ-ZAM-IB-2003-10

August 2003

(letzte Änderung: 22.08.2003)

(*) Modelling, Algorithms, and Informatics Group, CCS-3 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Table of Contents

Importing alternative paradigms into modern object-oriented languages.

Andrey V. Stolyarov

Moscow State Lomonosov University,
dept. of Computational Math. and Cybernetics,
MGU, II uch. korp., komn.747, Leninskie Gory,
Moscow, 119899, Russia

Abstract. The paper is devoted to the problem of importing alternative paradigms into an imperative object-orented environment. Several known solutions of the problem are discussed with explanation of their drawbacks. Then a new solution is introduced.
The solution is based on the fact that programming paradigms developed within alternative languages such as Lisp, Prolog, Refal etc. are in fact independent from their respective languages (e.g., from their syntax). Each of these languages implements a certain algebra, which in fact creates the paradigms. It is possible to represent such an algebra with object-oriented technique and get the respective set of paradigms within the primary language. Together with syntactic capabilities of the primary language (such as overloading of standard arythmetic operation symbols) this results in possibility for a programmer to use alternative paradigms (such as Lisp programming) right within the primary language (C++ or Ada95). No changes to the primary language is needed, nor is it required to apply any additional preprocessing to the code; only the standard translator of the primary language is used. The only thing needed to use the explained approach is an appropriate library.
As an illustration, the paper describes a C++ class library named InteLib which currently has a practically usable implementation for Lisp programming, and experimental umplementations of Refal and a subset of Prolog.

1 Introduction

Different programming languages encourage a programmer to use different ways to imagine the program being developed, the environment (hardware, operating system, user etc) and their interaction. The simplest way is to imagine a computer either as such (processor, memory, i/o ports etc) or a some kind of virtual machine capable to perform certain set of operations, and a program as a sequence of instructions those explicitly specify what operations are to be performed. This way of thinking is known as "imperative programming".

Another technique, proposed in early 1960s [10], is to represent the program as a set of functions. Each function gets zero or more arguments and computes a result. A function may use other functions in computations, including using itself, directly or indirectly (so called recursive function calls). No side effects are allowed, that is, if all the arguments are known, one can replace a function call with it's result and get the same program. This is known as "pure functional programming" [7].

Several years later, "logic programming" [12] was proposed. In logic programming, the program is thought as a set of logic facts (axioms) used to test statements and find objects that satisfy the given conditions. Pure logic programming also disallows side effects.

Obviously, both pure functional and pure logic programming are not suitable for interactive programs because interactive program changes its environment at least reading from the input and writing to the output so it's required to have functions with side effects to create an interactive program. In contrast with these two styles, the technique of object-oriented programming appeared in the middle 1970s looks like being created specially for interactive software development. The program and its environment are represented with so-called "objects" – abstract "black boxes" capable to exchange messages and perform various actions in response to a message [2].

The notion of a *programming paradigm* is often used to refer to a particular system of programming abstractions. It is important to notice that the notion of a programming paradigm is significantly unformal. There is no well-established classification of programming paradigms though there were many attempts to give one (e.g., [13]). For example, the languages Lisp[15], Refal[17], Miranda[18] and Hope[7] are all usually taken as functional languages. However, they in fact have less in common than in differences. E.g., a Refal function is based on text matching against patterns and transforming in accordance to the rule for the pattern first matched. There's no such capability in Lisp. This makes it more convenient to perform lexical and syntactic analyses with Refal than with Lisp. Hope and Miranda are pure functional languages while Lisp has global variables and local lexical bindings changeable as a side effect of a function, etc.

Nowadays the union of imperative and object-oriented paradigms is the most popular in the software industry. The union is implemented by such languages as C++ [16], Ada95, Java, Delphi, Object Pascal etc.

Using different languages together within a single project leads to different serious problems so it is rare practice; most projects are single-language, and the langage is one of these imperative object-oriented languages listed above. In most cases it is inconvenient to implement a whole project in Lisp, Refal or Prolog. This in fact results in that these languages are rarely used at all despite that they are extremely suitable for some subtasks in almost any project.

2 Example of a project suitable for multiparadigm technique

As noted in [6], "We may encounter application domains which can be modeled best with only one paradigm. But there may be other domains which can be represented more adequately using multiple paradigms". Furthermore, in almost any large software project there are subtasks for which alternative paradigms are suitable.

Consider we have a database with complicated relationship within its components, and a user who needs a good interface to the database, preferably close to a natural language. First of all, we have to analyse the queries user types at his console (lexical analysis). Then we need to determine what do they mean (syntax analysis). Finally, we need to create and perform a query to the database and return its results to the user.

It takes significant time to write a lexical analyser in an imperative language such as C++. If we do it in Refal, however, the work's complexity reduces by tens of times.

Next, we need to do some more processing to prepare the query. We should determine what formulae do mean and probably perform some transformations, optimizations etc. It is hard to operate with symbolic formulae in C++, but in Lisp all the symbolic transformations are programmed simply.

Now we are ready to request and retrieve a result from the database. It's not a problem when the query is simple; for instance, if we've got a database storing personal data of some people, a request like "Give me a home phone number of Bob Johnson" after lexical analysis and syntax transformations is not a problem to perform. However, the user might ask for a thing much harder to calculate. For example: "Find me a female who has graduated in 1997 in the Moscow State University as a computer science person, then got married in 1998 with a male who speaks fluent English and is older than his wife by 3 years". It is possible to create a database that stores all the necessary data, but a request like this might force us to write a whole program to complete it. Please note that we know the conditions and the only problem is to find the solution which satisfies them all. Logic languages such as Prolog, Datalog etc. are suitable for this purpose [5].

It looks like a good idea to make data flow such as shown at fig. 1

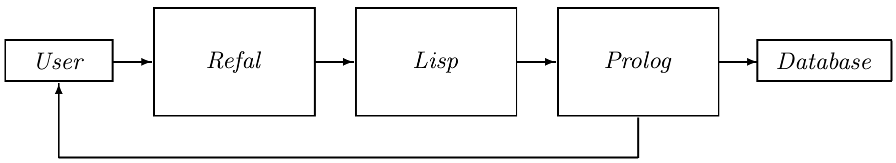

Fig. 1. Simple idea of data flow between parts implemented in different languages

However, the diagram at fig. 1 does not represent all the functionality of the hypotetical system. First, it must interact with a user, possibly via a network. Second, it should control the database, which is probably to be used by many users simultaneously. That leads to many technological problems (e.g., locking) to be solved by the system. In addition, we should not forget that the data is stored on physical (that is, *real*) disks, so we need to monitor the file system, check whether there's sufficient amount of free space, do some caching to increase performance etc.

Languages such as Refal, Prolog and Lisp are not good to do all these things. Their features are far from the real equipment capabilities. They are not so efficient as C++ and other "universal" imperative languages. For an artificial intelligence tasks such as the one described below, it is possible to trade efficiency of the software for development speed increase, but in a system task envolved to maintain the data storage it is inacceptable to loose in efficiency.

Besides that, interaction with a user in modern systems requires graphical interface (GUI), which is usually created as an event-driven system. It is inconvenient to create an event-driven system with an artificial intelligence language. Object-oriented languages such as SmallTalk[8], C++ or Java are much more suitable for this purpose.

The diagram at fig. 2 is closer to practice. We assume that "User Interface" and "Database Management System" are created with languages suitable for these purposes, probably C and C++. So we have C++ as a primary language and Lisp, Prolog and Refal as secondary languages.

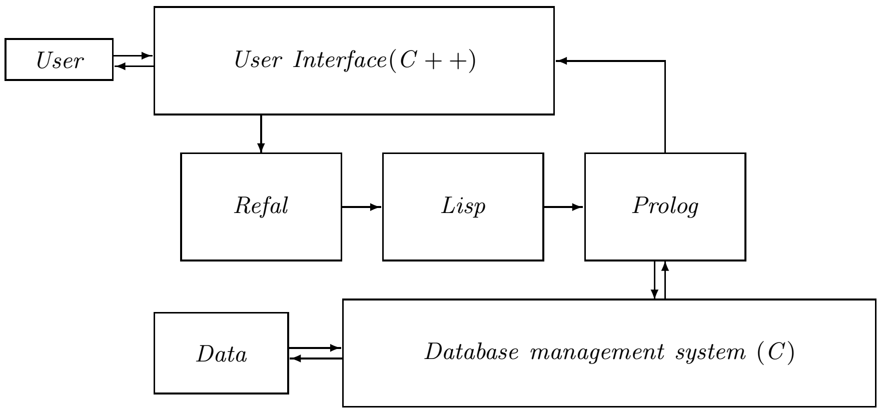

Fig. 2. Data flow closer to reality

However, when someone tries to use such an idea in a real project, she finds out it is much harder to implement such a system than to create a diagram. If we try to use an alternative language for a particular (small) subtask in a large project, we get into a trouble with integration of language tools that have so totally different nature (for instance, strictly typed imperative language as the primary language and a typeless functional language as the secondary language). There are problems in calling conventions, in sharing global data, in using heaps etc.

Furthermore, even the fact of using two or more different programming systems within one project makes the project harder to manage. If one of the programmers does not know one of the used programming systems, she could get into a trouble trying to build the project, to fix someone else's code etc. The difficulty of managing a project which uses two or more programming systems is so serious that this reason alone is able to prevent senior developers from making decisions of using different languages.

3 Different ways towards multiparadigm environment creation

It is obvious that an ability of using different paradigms together is attractive. There are certain difficulties though that prevent programmers from trying multiparadigm programming.

Before we introduce the new idea which hopefully allows to avoid most of the troubles, let us discuss some possible (and well-known) ways of creation of a multiparadigm environment.

We'll try to understand why each of them didn't become as widely-used as it is necessary to satisfy the need in multiparadigm programming. This will allow us to specify what do we actually want from the new technique.

3.1 Creation of another programming language

There were many attempts to create a new programming language for the purpose of multiparadigm programming (e.g., Leda[3], Oz[11] etc.). There is an unexpected trouble however. It is expensive to develop a new language. It is yet more expensive to bring the newly-developed language to the level of an industrial product so that it can be used in real software engineering practice, because this requires to support the language with useful software tools (compilers, debuggers etc.), as well as to create sufficient amount of documentation, tutorials and write and publish lots of books. But the real trouble is then to wait and see that the software engineering community doesn't tend to use the language in spite of all its advantages.

It is somewhat magic[1] when the community turns towards a new language, and this is a very rare kind of event, probably because the community is too conservative[2]. In fact it has to be. Really, starting to use a new technology requires to reeducate the personnel and change habitual methods of working. Both are very expensive, while outcome of changing languages is not so clear, specially to managers who make decisions.

This is why we decide not to try to create another language, even by extending an existing one. Enough of them are created already but that doesn't help.

[1] E.g., community preferred to use C++ with all its drawbacks while a similar but better-looking and carefully developed language Ada is in fact forgotten

[2] There's nothing bad though in this conservatism. Everything happens too fast in Computer Science so if the industry wasn't so conservative, we'd have nothing actually done.

3.2 A package of differently implemented programs

There are as well a few approaches to solve the problems of different programming languages integration within a single project. The simplest idea is just to write several programs, each in its own language, and make them interoperate using operating system's capabilities.

This kind of solution avoids problems with calling and data conventions, linking incompatibilities etc. It doesn't help tough with problems of using different programming systems within a single project.

Besides that, interoperation organization produces its own problems depending on a particular technology.

Using Unix style[3], when every program of the package has standard streams of input and output and the interoperation is done with command languages, we have to convert all the data somehow into a text representation, and then analyse the representation in another program of the package. Another drawback is that it is not always convenient to use the standard input/output streams for interoperation with another part of the package.

There are some attempts to make another, more convenient for a programmer standard way to organize interoperation of different programs, such as CORBA and COM. However, such technologies theyselves are complicated enough so that making a program to support them could be comparable in difficulty with solving the task the program is actually written for, and they are best handled with object-orientred languages, producing troubles with functional or logic languages.

3.3 Embedded interpreters

Another solution is to build an interpreter of a secondary language into the primary language. In the simplest case the interpreter is implemented as a module, which has an appropriate interface. The main program (e.g., written in C++) feeds the interpreter with the text of a module created in the secondary language (e.g., Lisp), then passes the initial data, runs the interpreter and reads the results back.

One of more advanced techniques is known as 'embeddance' of one language into another. In this case the primary language is enlarged with some certain constructs which allow to insert constructs of a secondary language into a code in the primary language. In this case we need a preprocessor which handles embeddance constructs and produces a plain code in the primary language which is then compiled in a regular way. In fact such a preprocessor replaces a foreign construct with some code which converts all necessary variables into a text, creates an appropriate query and then passes it to an interpreter, and then converts the received result as necessary. This technique is successfully applied to the case of writing database operation software using SQL-based database management system.

Such a solution, though, has many disadvantages:

[3] Unix style is mentioned as a multiparadigm environment in, e.g., [14]

- The solution does not allow us to call primary language functions from within the interpreted code. Only primary language functions can call the secondary language code, but not vice versa.
- The secondary language code is fully interpreted. That is, when the program runs, every call to the interpreted language is given in its text representation. The embedded interpreter has to perform lexical and syntactic analyses "on-the-fly" which may lead to efficiency losses.
- "The last but not least" – the results of the interpreted code calling are also presented in text form, so we need to analyse it in the main program. Just remember that analysing of strings is one of the tasks we want to avoid in C++ code and perform with an artificial intelligence language, preferably Refal, and you'll realize that something is wrong.

Besides that, mixing up interpreted and compiled execution within one program doesn't look like a fair solution anyway. It is clear that we can't avoid interpretation completely for such languages as Lisp or Prolog, but at least we could expect there will be no lexical and syntactic analyses at runtime.

3.4 Extendable interpreters

The opposite solution is to choose an interpreted language as the primary one and provide mechanisms to extend the interpreter with functions implemented in another language (usually the language in which the interpreter is initially implemented). One of the well-known examples is Tcl. Its interpreter allows to call C code which is compiled into a shared library following certain simple calling conventions.

The main drawback of this method is that the primary language **must** be interpreted which may be inappropriate in some cases.

3.5 Cross-language linkage

Having certain amount of patience, it is possible to compile and link modules written in different languages together. As it was mentioned before, this produces numerous problems with differences in calling conventions, data representation conventions etc. Furthermore, it almost always requires to make changes to the existing programming systems (for example, to reimplement compilers so that they could produce compatible object modules). As of practice, all these hardships are sufficient to prevent programmers from trying multiparadigm programming. And, anyway, we don't reach real integration of languages this way because the languages theyselves are not designed for multilanguage environment (e.g., there's a problem how to call a C function from Lisp code – how to *specify* such a call using Lisp syntax).

3.6 Compilation from one language into another

The difficulties of linking together modules implemented in two defferent languages can be reduced if one of the languages is first compiled into the other.

Many of Scheme translators actually produce C code which can then be compiled in usual manner. Thus there's no problem to link such a code with some modules implemented in C and/or C++.

We still don't know how to specify a C function call in Scheme syntax, that is, only Scheme functions can be called from C, but not vice-versa. Also, the programmer needs to understand the internal data structures of the particular implementation of Scheme in order to compose a call to a Scheme function and/or analyse the results. This is inappropriate because internal data structures are usually not well-documented.

3.7 Paradigms without a language

There's also a chance to brainstorm why do we actually want to use another language for a particular subtask, that is, what features does it have the primary language doesn't provide, and then just implement them (e.g., as a library).

Consider we use C++ as the primary language and we for some reason we feel it useful to have heterogenous lists[4] as we do in Lisp. It is possible to implement them with C++ template classes. First, we create a base class which implements the common behaviour of all items of such a list (e.g., a pointer to the next item, a pure virtual function which returns the size of this object etc.) Having this class, we define a template child of it. The template gets the type of the stored value as its parameter. Each item of such a list would be an instance of the template. Using C++ runtime type identification (RTTI) we can tell one type of an item from another when the list is handled.

Practice shows however this doesn't *completely* satisfy the programmers' needs. Each language grows a special environment where new techniques and methods appear, and these methods usually base on more than one paradigm (such as heterogenous lists). If we remember Lisp working on a C++ project, we probably won't stop with Lisp lists alone. Once we implemented the lists, the next thing we might need is Lisp mapping functions, or Lisp destructive list changing and garbage collection, and so on.

3.8 Summary

Now let's summarize what conditions do we want to meet with a new solution. We need a framework for multiparadigm programming which

1. allows to use one of the languages widely accepted by the industry as it is, i.e. with no changes to the existing compilers and other tools;
2. delivers additional paradigms as they exist in the choosen alternative language, all (or almost all) together;
3. doesn't place any limitations over interparadigm function calls and sharing data between different code;
4. doesn't require lexical and syntactic analyses of any parts of the code at runtime.

[4] Each element of the list may have its own type

In the next chapter we introduce a technique which complies the above requirements. It is discussed assuming C++ is the primary language and Lisp is the language we need to import the set of paradigms from.

4 The key idea

In order to explain the idea of a new technique, let's discuss what do we actually need from the secondary language (e.g., Lisp). Do we, for example, need its syntax? Perhaps we don't. Generally speaking, **we need the paradigms developed around the language, not the language itself**.

Lisp language implements a kind of algebra on so called S-expressions. Both program and data are built of S-expressions. The basic operations on S-expressions are:

– composition (allows to make a list or an arbitrary binary tree of S-expressions)
– decomposition (retriving elements of a list or a tree)
– evaluation (allows to treat an S-expression as a code and perform the according operations)
– lambda (allows to build an S-expression of a functional type, so-called closure, with a given list of formal parameters and a list representing the function's body)

Special type of S-expression called *symbol* (in the terminology tradition to Common Lisp [15]) or *identifier* (in the Scheme's terminology [9]) has additional operations - assigning a value, binding a value and assigning a function. In real dialects of Lisp this set is wider, but we'll limit to these 3 operations.

Besides that, there are additional basic operations (that is, operations which require one to know the internal representation of S-expressions in order to implement such an operation). Some of them are necessary to make the algebra useful (e.g. arythmetic operations on numberic S-expressions), while others are intended for the user's convenience.

The mentioned operations create an algebra on the space of S-expressions. We will denote the introduced algebra as *S-algebra*.

It is clear that S-algebra being implemented in any particular way will give us the full set of Lisp paradigms. S-algebra may be implemented without an actual Lisp interpreter. All we need is to keep it useful, that is, provide a convenient instrumental basis to operate S-expressions and apply all the necessary operations.

In some languages (including C++ and Ada) it is possible to overload standard operations such as `+`, `-`, `/` etc. This allows to implement an arbitrary algebra using very natural and convenient syntax. For example, one can implement a mathematical notion of a vector using `+` for vector addition, `-` for vector difference operation, `*` for the scalar multiplication and (for instance) `^` for vector multiplication.

In the same manner we can implement the notion of S-expression with a class (or, more precisely, with a polymorphic hierarchy of classes) and invent a certain set of operations so as to implement the whole S-algebra presented in Lisp.

5 Representation of S-algebra with C++

In this section the architecture and design of a C++ class library named InteLib [1] is explained as an illustration of the proposed idea.

5.1 S-expressions of various types

The notion of S-expression is represented with an abstract class which for historical reasons is called LTerm. In Lisp, there are S-expressions of different types (numberic constants, string constants, symbols, dotted pairs, functional objects/closures etc.) A polymorphic inheritance technique is used to represent differend types of S-expressions. The LTerm hierarchy is shown at fig. 3.

In the discussed version of the library the `LTerm` class' children serve to represent various types of S-expressions:

- `LTermInteger` and `LTermFloat` represent numberic constants;
- `LTermString` represents string constants;
- `LTermLabel` is introduced to represent S-expressions whose role in the system is determined by the particular instance of the object (such as Common Lisp symbols, as well as `#t` and `#f` in Scheme);
- `LTermSymbol` represents Lisp symbols;
- `LDotPair` represents dotted pairs which Lisp lists are built of;
- `LForm` represents generic functional S-expression such as library function, user-defined function or lexical closure, Lisp macro etc.
- several additional classes to represent miscellaneous features such as hash table, i/o stream etc.

InteLib supports two types of numberic constants, namely integers and floats. Compile-time options of the library allows to choose what numberic types we actually need. It is possible to cause `LTermInteger` to use `short`, `int`, `long` or `long long` type to store the actual value, as well as `LTermFloat` can be tuned to use `float`, `double` or `long double` form of a floating point number.

Strings are implemented assuming a string constant itself is never changed. In contrast with Common Lisp, there is no vector type of S-expression in the discussed model so a string is considered as just an atomic value.

There is no special type of S-expression for single characters. They are represented with an `LTermString` object as a string which has length of 1.

Lisp symbols are implemented with class `LTermSymbol`. The class is capable to hold a reference to an object which represents the current dynamic value of the symbol and to another object which represents the function associated with the symbol. Stuff related to lexical bindings of a symbol is implemented outside

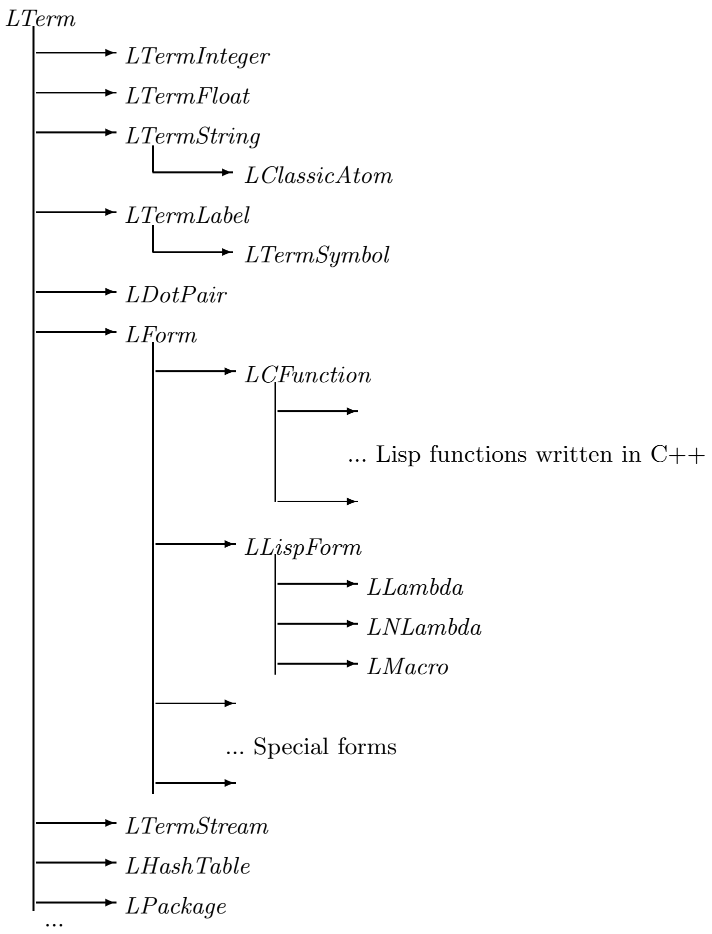

Fig. 3. LTerm classes hierarchy intended to represent S-expressions of various types

the class but is supported with its methods (those related to setting and getting the value).

The notion of an empty list may be implemented by any object of S-expression; the only condition is that the address of the object is known at the compile time because the end-of-list check is performed just by comparing pointers. Usually an object of the class `LTermSymbol` (for dialects close to Common Lisp) or LTermLabel (for Scheme-like dialects) are used for this purpose.

To represent functionals as data, `LForm` subhierarchy has been developed within the `LTerm` hierarchy. The subhierarchy has `LCFunction` and `LLispForm` classes derived directly from `LForm`. Lisp special forms are also represented with classes descended directly from `LForm`.

The `LCFunction` class represents functions implemented in C++, including all "built-in" functions such as `CAR`, `CDR`, `CONS` etc. To add a new Lisp function to the library, a programmer needs to declare a child of `LCFunction` with only one method (`DoCall`) overriden in order to implement the necessary functionality.

The `LLispForm` class is intended to represent forms defined with Lisp constructions (lambda functions, nlambda functions and macros). It has references to the lambda-list (the list of formal parameters), the function body and the lexical context[5] of the form. `LLispForm` class has child classes corresponding to different kinds of forms:

– `LLambda` (ordinary Lisp function which evaluates all its arguments and then evaluates the body in its own context);
– `LMacro` (Lisp macro evaluated as in Common Lisp);
– `LNLambda` (Lisp function which does not evaluate its arguments).

5.2 Garbage collection

Some of `LTerm`'s children are large so that it is inefficient to pass them by value. However, sometimes these objects are constructed within a function which may be called for the value as well as for its side effect so that it is no good to return the created object by pointer (if the function is called for the side effect then the constructed object goes to garbage).

In order to provide garbage collection, another class (called `LReference`) is provided. It has precisely the same size as a pointer do so that it is not so bad to pass it by value. An object of `LReference` class acts just like a pointer to an `LTerm` object having necessary operations including `*` and `->`. `LReference` is intended to be the primary interface to the library. It has a lot of constructors which allow to construct an S-expression from a value of any base C++ data type (integers, floats, strings etc).

In most cases, the objects of `LTerm` class hierarchy reside in dynamic memory and are not operated directly (altough it is possible). `LReference` objects are used to handle `LTerms`.

Besides other features, `LReference` notifies the pointed object when another pointer to it is created or an existing pointer is no longer pointing to it (e.g., it is assigned with another value or destructed). `LTerm` class performs simple reference counting and deletes the object once it has zero references.

Reference counting is choosen for its simplicity. It has well-known problems (including the problem of cyclic constructions). If it is inappropriate for a particular application to use reference counting, then any of existing C++ garbage collection libraries can be used instead. The library has a compile-time option to switch off the reference counting code.

5.3 Lexical bindings

There are also additional classes that represent (in traditional Lisp terminology) a notion of lexical context. Objects of these classes are not operated by user in

[5] The notion of lexical context is implemented with a separate class LLexicalContext

Table 1. Examples of Lisp expressions representation with C++ constructs

C++ constructs	Lisp equivalent
`(L| 25, 36, 49)`	`(25 36 49)`
`(L| "I am the walrus", 1965)`	`("I am the walrus" 1965)`
`(L| 1, 2, (L| 3, 4), 5, 6)`	`(1 2 (3 4) 5 6)`
`(L| (L| 1, 2), 3, 4)`	`((1 2) 3 4)`
`(L| MEMBER, 1, ~(L| 1, 3, 5))`	`(member 1 '(1 3 5))`
`(L| APPEND, ~(L| 10, 20), ~(L| 30, 40))`	`(append '(10 20) '(30 40))`
`(L| 1 || 2)`	`(1 . 2)`
`((L| 1, 2, 3)|| 4)`	`(1 2 3 . 4)`

most cases. The library does not, however, hide them from the user because in some cases it might be useful to create a context manually. There's always one active lexical context (possibly special null context). In order to use a context it must be activated. Then it is affected by operation of binding a value to a symbol. The context itself affects operations of assignment and retrieving a value of a symbol.

5.4 List composition operations

In Lisp there's an operation of constructing a list of an arbitrary length denoted by parentheses. The operation has variable 'arity'. For example, a construct `(1 2 3)` has 3 arguments and builds a list of 3 items - 1, 2 and 3. Besides that, there's a binary operation which builds a dotted pair, such as `(1 . 2)`. The construct `(1 2 3)` has the same effect as a superposition of 3 dotted pair constructors, like this:

```
(1 2 3)    ==    (1 . (2 . (3 . NIL)))
```

We can implement an operation similar to the Lisp's `( . )` and it will allow us to build any list of S-expressions. It is though a bit inconvenient to create lists using this operation only (imagine you couldn't use plain lists in Lisp, only dotted pairs).

Another problem is that one might want to use just a C++ constant or expression without explicit cast of it to an S-expression, e.g. '3', not a construct like `LReference(3)`, so in case of an overloaded standard operation we need at least one of operands already of LReference type, which allow a C++ compiler to understand we mean the overloaded operation, not a standard one. This requirement also prevents us from using C++ functions with unspecified number of arguments because there's no way to determine at run time what types of actual parameters do we have.

The problem is solved replacing the Lisp '( )' operation of an arbitrary list composition with two operations. First of them creates a list of one element, while second adds an element to a given list. It is clear the two operations allow to create an arbitrary list, that is, their combination has the same functionality as the Lisp '( )' operation.

The first operation always has exactly one argument, but we can't use a symbol of any standard unary operation for it because we want it to be applicable to an expression of any standard type. We also don't want to use a plain function for this purpose because parentheses would make our constructs less clear. The problem is solved with a class `LListConstructor`, which is created to be a label for a binary operation to show the compiler to apply an overloaded one instead of the built-in operation. Usually there's only one instance of `LListConstructor` named `L`. For example, an operator `L|3` returns an `LReference` object that represents Lisp construct `(3)`.

For appending a new item to a list we can overload any overloadable binary operator. For a better clarity we decide to use C++ comma (`,`) for this purpose. Left-hand operand of a comma is always an LReference representing a list. Comma destructively changes the list replacing the final `NIL` with a dotted pair of `(X . NIL)` where `X` is its right-hand operand casted to an `LReference`. This makes it possible to represent Lisp lists in C++ as shown in table 1.

There's a supplementary unary operation in most of Lisp dialects which allows to construct a list of two elements, first of which is a symbol `QUOTE` while the second is the operation's operand. The operation is usually denoted by a single quote symbol (`'`). InteLib overloads the operator `~` (tilde) for this purpose. See table 1 for an example[6].

For composing dotted pairs and dotted lists InteLib offers an operation `||`. The left-side operand must be a list (possibly of one element). The operand appearing at the right side of the operator is converted to LReference and then the operation replaces the last `NIL` of the given list with whatever it constructed from the right-side operand. See table 1 for an example.

Note the parentheses in the last example. They appear because the comma operator has lower precedence than `||`.

5.5 Operations implemented as regular methods

The most important operation on S-expressions is the *evaluation* of an S-expression. Evaluation is an unary function which maps from $S_x \backslash S_{ux}$ to S_x where S_x is a space of all possible S-expressions and S_{ux} is a set of 'unevaluable' S-expressions (such as closures). Performing evaluation of an S-expression one can also get a side effect.

All constants evaluate to themselves with no side effects. Variables evaluate to their values, if any. Evaluation of an unbound variable generates an error.

Evaluation of a list interpretes the first element as an instruction what functional object to apply to the rest of the list as a list of parameters. In most cases, the resting elements of the list are evaluated and the appropriate function is applied to a list built of the results. The first element of the list must be either a symbol which has an associated functional object or a Lambda-list.

[6] The operator is oveloaded for LReference class so it is possible to apply it to a list or a Lisp symbol, but one can't use it with strings or numberic constants. It is unnecessary anyway to quote them since they always evaluate to themselves

It looks like we need to implement two operations in order to support the evaluation: the evaluation itself (as a polymorphic method of the `LTerm` class) and an operation of *application* of a functional S-expression (a one that belongs to `LForm` subhierarchy) to a list of parameters. The two operations are implemented as methods called '`Evaluate`' and '`Call`', respectively. The '`Call`' method generates an error when called for an object of a non-functional type. '`Evaluate`' generates an error when it is impossible to evaluate the given S-expression. The `IsSelfEvaluated` method allows to determine whether the object represents a constant that always evaluates to itself.

Besides that, there are methods '`Car`' and '`Cdr`' in LTerm class. They return the respective cells of a dotted pair when called for an `LDotPair` object. For an object that represents empty list both methods return empty list. Calling these methods for a non-list object will cause an error.

Another important method named `TextRepresentation` allows to create a human-readable representation of any given S-expression.

It is well known that there are 3 different predicates of equality in Lisp, called `EQ`, `EQL` and `EQUAL`. `EQ` predicate is the simplest one, it just compares two addresses. `EQL` is a bit more flexible. Two objects may be not the same while representing the same value (e.g. two instances of an integer constant 2). In order to make it possible to implement `EQL` predicate for any `LTerm` object there's a virtual method `SpecificEql` which returns false by default. Implementation of `EQL` checks for equality of addresses first, and only if the objects are not `EQ`, it calls `SpecificEql` for one of them passing the other as an argument, so that it doesn't cause a misbehaviour when `SpecificEql` returns false when the compating objects are the same, that is, equal in the sence of `EQ`.

Another important operation, Lambda, is implemented by a constructor of `LLambda` class. For example, expression

```
LReference(new LLambda(NULL, (L| A),
                       (L| (L| PLUS, A, 1))))
```

creates a closure with `NULL` lexical context. The closure takes a numberic S-expression and returns a number which is greater by one. In Lisp such a closure would be represented as `(lambda (a) (plus a 1))`. In order to create a real closure that has lexically bound variables, the appropriate lexical context must be passes as the first argument of the `LLambda` constructor.

There are other methods in classes of the library which a provided mostly for user's convenience. They include typecasting operations, which allow to convert a constant S-expression into a base C++ value. For example,

```
LReference(3)->GetInteger();
```

will return 3, while

```
LReference("Hello world")->GetString();
```

will return a pointer to a constant string `"Hello world"`. Calling such a method for a wrong type of S-expression causes an error.

5.6 Operations performed by standard Lisp functions

Standard, or built-in, functions play a key role in Lisp functionality providing a basis for building programs. They can also be thought as operations on S-expressions, that is, as elements of S-algebra.

In Lisp there are symbols that initially have associated built-in functions. InteLib doesn't provide such symbols in order to allow a user to use whatever names she wants for these symbols. The functional objects representing well-known Lisp functions are direct children of `LCFunction` class (for functions that evaluate all arguments) or of `LForm` class (for special forms). They usually have names such as `LFunctionCar`, `LFunctionCons`, `LFunctionLet`, `LFunctionDefun` and so on.

For convenience there is a generic class

```
template<class F> class LFunctionalSymbol
```

whose argument must be a class that represents a particular function. An instance of `LFunctionalSymbol` differs from `LSymbol` in that its constructor creates an object of the given finctional class and lets it be the associated function of the symbol. For example, one might want to add the following declaration to the program:

```
LFunctionalSymbol<LFunctionCar> CAR("CAR");
LFunctionalSymbol<LFunctionCdr> CDR("CDR");
LFunctionalSymbol<LFunctionCons> CONS("CONS");
LFunctionalSymbol<LFunctionCond> COND("COND");
LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
```

etc. As usual, the constructor's argument sets the textual name of the symbol which is used by `TextRepresentation` method.

Consider, for example, the following Lisp code:

```
(defun isomorphic (tree1 tree2)
 (cond ((atom tree1) (atom tree2))
       ((atom tree2) NIL)
       (t (and (isomorphic (car tree1)
                           (car tree2))
               (isomorphic (cdr tree1)
                           (cdr tree2))))))
```

One can write the module shown at fig. 4 to do the same thing in C++. The module compiles with an ordinary C++ compiler without any additional preprocessing. The symbol `ISOMORPHIC` is public and can therefore be used in other modules.

Please note there are no definitions of symbols `T` and `NIL`. They are provided by the library as well as symbols `QUOTE` and `LAMBDA`. The library needs symbols `T`, `NIL` and `QUOTE` because it is possible to obtain them from certain operations without mentioning them in a program. Consider the following example:

```
//       File isomorph.cpp
#include "intelib.h"
LSymbol ISOMORPHIC("ISOMORPHIC");
void LispInit_isomorphic() {
  static LSymbol TREE1("TREE1");
  static LSymbol TREE2("TREE2");
  static LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
  static LFunctionalSymbol<LFunctionCond> COND("COND");
  static LFunctionalSymbol<LFunctionAtom> ATOM("ATOM");
  static LFunctionalSymbol<LFunctionAnd> AND("AND");
  static LFunctionalSymbol<LFunctionCar> CAR("CAR");
  static LFunctionalSymbol<LFunctionCdr> CDR("CDR");
  (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
    (L|COND,
      (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
        (L|(L|ATOM, TREE2), NIL),
        (L|T, (L|AND,
          (L|ISOMORPHIC, (L|CAR, TREE1), (L|CAR, TREE2)),
          (L|ISOMORPHIC, (L|CDR, TREE1), (L|CDR, TREE2)))))).Evaluate();
}
//      end of file
```

Fig. 4. Example of a C++ module that defines a function in a manner of Lisp

```
(eql 1 2)   ->  nil
(eql 1 1)   ->  t
(car ''a)   ->  quote
```

The symbol LAMBDA can't be obtained in such a way, but it has special meaning regardless of it's possible value and/or associated function, so we have to rely on the symbol itself (that is, for example, on the object's address). That's why these 4 symbols are provided by the library in contrast with all the other well-known symbols.

It is important to understand that there's no Lisp as such in the C++ module shown at the fig. 4. The module is written in C++ language. The compiler knows nothing about special meaning of all these commas and vertical bars; they are handled as functions just like in any program which overloads standard operators. Thus we can say we made no changes to the primary language, at the same time allowing a programmer to use paradigms from another (secondary) language. Only paradigms are imported from Lisp, not the language itself. In other words, we **import S-algebra which brings us the paradigms of Lisp without Lisp language as such.**

6 Translation from Lisp to C++

The primary goal of the InteLib library is to bring Lisp paradigms to C++. However, as a side effect it opens a clear way to translation of Lisp code into

C++. The translator is made which uses simple rules of transformation of the code. Besides that, the translator generates the necessary definitions of symbols and performs some other tasks as to allow using several Lisp modules within a single project. Since the translator is not the main goal of the project, we don't explain it in details in this paper; we only give a short description.

The translator takes one or more Lisp files and proudces a C++ module (that is, a "source" file and a header file). The translator understands a certain set of so-called translation directives (top level forms beginning with a token `%%%`), which allow to control how names are translated etc.

The character set for C++ identifiers differs from the traditional one for Lisp symbols. C++ identifiers are case-sensitive and can consist of letters, digits and the underline symbol. The first character of identifier must be a letter or the underline, not a digit. Lisp symbols are case insensitive and may be built of letters, digits and various symols such as `+`, `-`, `*`, `_`, `%` etc. so we need a certain translation algorythm.

The fact that Lisp symbols are case sensitive and C++ identifiers are not is very helpful in translation of names from Lisp to C++. This allows us to bring all letters in a Lisp symbol to the upper case and leave the lower-case letters to represent all these pluses, dashes and other symbols that are not allowed in C++ identifiers. For instance, the Lisp symbol `read-char` might be represented as `READdashCHAR`. If a symbol's name begins from a digit (which is illegal in C++), we prepend it with a lowercase letter, for instance - "`d`". The symbol `7seas` having been translated this way becomes the identifier `d7SEAS`.

There are some exceptions from the general rules. For example, the Lisp symbol `null` would be translated to `NULL` producing a name conflict with some of the standard header files. That's why it is translated as `lNULL` (as specified by an appropriate translation directive). User can add her own exceptions, rules etc.

As to experience, using of the translator is good when a whole module is inplemented in Lisp because the traditional Lisp syntax is more convenient than the introduced C++ implementation of S-algebra. It is still possible, however, to avoid using the translator.

The dialect of Lisp recognized by the translator was named InteLib Lisp. It is a very short and simple dialect designed keeping in mind that it is to be used as a secondary language. It ommits many features of modern Lisp dialects because they are considered not to be essential.

7 Logic programming

The explained technique can be applied to secondary languages other than Lisp. One of the obvious ways of further development is to apply it to one of the logic programming languages.

As of now, an attempt is done to apply the technique to a very restrictive subset of Prolog[4]. The Prolog part of the library, unlike the Lisp part, has primarily a demonstration value; creating a library useful in real programming practice is the subject of further work.

The implemented subset of Prolog has no dynamic data structures (lists and functors), just like in Datalog[5]. Unlike Datalog, the implemented dialect has the *cut operation*.

Prolog machines operate on data which is similar to S-expressions (in fact, the only difference is functors). Creating a model of a Prolog machine, a decision was made to reuse the classes already implemented to represent Lisp data[7].

To represent the notion of *predicate*, `DlAbstractPredicate` class is invented. It has a pure abstract method named `DlCreateAbstractIterator` which is intended to create an iterator to fetch, one by one, solutions of a given predicate provided with the appropriate number of arguments.

To create atoms of given predicate (e.g., having predicate `father`, create the atom `father(john, X)`), the class provides `operator()` for 0, 1, 2, ..., 10 arguments of the type LReference[8].

Another class, derived from `DlAbstractPredicate` and named `DlPredicate`, represents the predicate which is the part of Prolog program (that is, a predicate formed of clauses of goals).

Prolog atoms (constructs such as `father(john, mary)`, `vertex(X)`, etc.) are represented with the `DlAtom` class which is in fact just a pair of a predicate and an argument list. Atoms are usually created within functions, locally, so another smart pointer is necessary. It is named `DlAtomRef`. This smart pointer is used as the primary interface to the `DlAtom` class. One of the most important operations of `DlAtomRef` is `operator<<=()` which is used instead of the well-known Prolog symbol `:-`. The operator adds another clause to the appropriate predicate and returns a reference to the object of the class `DlPredicate::DlClause`. That object, in turn, has `operator,()` to add goals (atoms) to it.

Prolog variables are represented using class `DlVariable`.

Using all these classes and operations, we can represent the Prolog clause

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

with C++ expression

```
grandfather(X, Y) <<= father(X, Z), father(Z, Y);
```

where `X`, `Y` and `Z` are objects of the class `DlVariable`. Facts such as

```
father(john, george).
```

are represented using another form of the operator `<<=`:

```
father(john, george) <<= true;
```

[7] So the dialect in fact can handle lists, but it still treats them as atomic data values, e.g., when doing unification

[8] There is no operator with variable parameters list, because LReference objects, being objects of a class, can't be passed through ... in C++.

Prolog code example:

```
father(john, george).
father(george, alex).
father(alex, alan).
father(alan, poul).
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Equal C++ code:

```
#include "il_dlog.h"
LSymbol john("john"), george("george"),
  alex("alex"), alan("alan"), poul("poul");
DlVariable X("X"), Y("Y"), Z("Z");
DlPredicate father, grandfather;
void PrologInit_father_grandfather() {
    father(john, george) <<= true;
    father(george, alex) <<= true;
    father(alex, alan) <<= true;
    father(alan, poul) <<= true;
    grandfather(X, Y) <<= father(X, Z), father(Z, Y);
}
```

Fig. 5. Prolog-like C++ code example

Fig. 5 shows an example of a Prolog-like C++ code which uses the predicates `father` and `grandfather`. Now, the code

```
iter = grandfather(X, Y).CreateIterator();
bool rc;
do {
  PDlSubstitution solution;
  rc = iter->NextSolution(solution);
  if (rc) {
    printf("%s\n", solution->TextRepresentation().c_str());
  }
} while (rc);
```

will print the following solutions list:

```
{X/john Y/alex}
{X/george Y/alan}
{X/alex Y/poul}
```

As we already noted before, the implemented model doesn't do unification of dynamic data structures though Lisp lists can be operated with, considering them atomic datums. To produce a more interesting demo, let's add a built-in predicate named `DLCONS(car, cdr, cons)`. The predicate is implemented by another class derived from `DlAbstractPredicate`, named `DlPredicateCons`.

The predicate is able to work having any of its arguments specified or unspecified (that is, a variable is given instead of a value).

Another useful predicate is `DlPredicateLispcall` (to call the Lisp machine explained before).

The object `DlCut` is a special value of `DlAtomRef` which represents the cut operator.

Note also that a `DlPredicate` without any clauses always fails, so to implement an *always failing* goal we can just create an empty `DlPredicate`.

Having all these objects, we are ready to write a simple program which finds a path in a given graph (fig. 6).

Now, if we create the appropriate iterator with

```
iter = Shortpath(2, 4, X, 2).CreateIterator();
```

the solution finding code like the one shown above will print the solution

```
{X/(2 1 4)}
```

Unlike the Lisp part of InteLib which is already useful in some practical cases, the explained Prolog part is only a simple demo. It is planned to implement a more practically useful library in the close future.

8 Conclusions

The most important advantage of the proposed technique is that there's no need for two programming systems within a project. The existing C++ compiler is always used, and the only thing required to use the technique is a C++ class library which has a relatively simple imterface.

It is also possible to use another primary language. The only requirement to it is the possibility of overloading of standard operations. In particular, Ada95 may be used as the primary language as well (at least for modelling Lisp as the secondary language). Implementation of the appropriate library for Ada95 might be one of the further work goals.

References

[1] E. Bolshakova and A. Stolyarov. Building functional techniques into an object-oriented system. In *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam.

[2] G. Booch. *Object-oriented Analyses and Design*. Addison-Wesley, Reading, Massachusets, second edition, 1994.

[3] T. A. Budd. *Multy-Paradigm Programming in LEDA*. Addison-Wesley, Reading, Massachusets, 1995.

[4] A. Calmerauer, H. Kanoui, and M. van Caneghem. Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2(4):271–311, 1983.

```
LListConstructor L;
DlPredicateCons DLCONS;
DlPredicateLispcall DLLISPCALL;
DlAbstractPredicateIterator *iter;
DlVariable X("X");
DlVariable Y("Y");
DlVariable Z("Z");
DlVariable P("P");
DlVariable N("N");
DlVariable V1("V1");
DlVariable V2("V2");
DlVariable V3("V3");
DlPredicate Edge("Edge");
DlPredicate Edge2("Edge2");
DlPredicate Member("Member");
DlPredicate Shortpath("Shortpath");
DlPredicate Fail("Fail");
Member(X, Y) <<= DLCONS(X, V2, Y);
Member(X, Y) <<= DLCONS(V1, V2, Y), Member(X, V2);
Edge(1, 2) <<= true;
Edge(1, 3) <<= true;
Edge(1, 4) <<= true;
Edge(1, 5) <<= true;
Edge(5, 3) <<= true;
Edge2(X, Y) <<= Edge(X, Y);
Edge2(X, Y) <<= Edge(Y, X);
Shortpath(X, Y, P, 1) <<=
             DlCut,
             Edge2(X, Y),
             DLCONS(Y, L, V1),
             DLCONS(X, V1, P);
Shortpath(X, Y, P, N) <<=
             DLLISPCALL((L|lt, N, 1), T),
             DlCut,
             Fail();
Shortpath(X, Y, P, N) <<=
             DLLISPCALL((L|minus, N, 1), V1),
             Shortpath(Z, Y, V2, V1),
             Edge2(X, Z),
             DLCONS(X, V2, P);
```

Fig. 6. Graph path finding program

[5] S. Ceri, G. Gottlob, and L. Tanka. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[6] U. W. Eisenecker. Future trends in multi-paradigm programming. Position Paper for the ECOOP'98 Panel on Multi-Paradigm Programming, 1998.

[7] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Reading, Massachusets, 1998.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusets, 1983.

[9] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on Algorithmic Language Scheme, 1998.

[10] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[11] M. Müller, T. Müller, and P. Van Roy. Multiparadigm programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Workshop on the Future of Logic Programming*. International Logic Programming Symposium, 1995.

[12] J. Robinson. Logic programming - past, present and future. *New Generation Computing*, 1:107–121, 1983.

[13] D. Spinellis, S. Drossoupoulou, and S. Eisenbach. Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming. In J. Gutknecht, editor, *Programming Languages and System Architectures International Conference*, volume 782 of *Lecture Notes in Computer Science*, pages 191–207, Zurich, Switzerland, March 1994. Springer-Verlag.

[14] D. D. Spinellis. *Programming paradigms as object classes: a structuring mechanism for multiparadigm programming*. PhD thesis, University of London, London SW7 2BZ, United Kingdom, February 1994.

[15] G. L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusets, third edition, 1997.

[17] V. Turchin. *REFAL-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, 1989.

[18] D. A. Turner. Miranda – a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Proceedings of the Conference of Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, 1985. Springer-Verlag.