Thorn language: a flexible tool for code generation

Yuri Okulovsky

Ural State University
Yekaterinburg, Lenina str. 51
Email: yuri.okulovsky@gmail.com

***Abstract*—This paper presents a new approach to domain-specific languages creation. Instead of defining both language syntax and semantics for each case, the same general-purposed markup language Thorn is used. The original model of translation associates commands inside a Thorn document with programs written in some script language. When the compiler needs to execute a command, it launches a corresponding program, passes data from the document to the program, and uses an output value of the program as the result of the command. We describe an approach to code generation based on Thorn, and compare the approach to other known code generation methods. We give various examples of Thorn-based code generators.**

***Index Terms*—code generation, domain-specific languages, automated programming, language-oriented approach**

I. Introduction

Every programmer likes writing elegant code, implementing sophisticated algorithms and developing original architecture. Real software, however, often consists of stereotyped and uncreative code: business logic control, layout of widgets, interaction with database, automata dispatch tables, etc. The more features a software product has, the larger amount of stereotyped code it contains. This leads to the staff expansion and project management issues. In addition, a comprehensive testing of stereotyped code is required. The natural desire emerges to eliminate these inconviniences.

In fact, all the history of programming languages is a history of attempts to decrease the percentage of stereotyped code and to improve its structure. Simple arithmetical operations in Assembler require several lines of code. More modern languages like C allow writing the same operations in one line. Memory management in C demands special attention, leading to lots of stereotyped code and numerous memory leaks. These issues was resolved with garbage collection in Java and C#. The most recent developements in programming languages simplify significantly collection management and interaction with databases (LINQ [13]), widgets layout (WPF [13]), or contracts control (Eifell [10]). However, software products often have unique patterns in addition to well-known ones. These patterns are typically not supported by languages.

The promising approach to described problems is code generation: writing programs that write programs [8], [5]. The simpliest case is a tool that allows description of the desired code in text or graphic format, and then produces the code. For example, Microsoft Visual Studio contains a special wizard for visual creation of widgets and forms [2]. Another example is production of parsers from grammars [9]. There are many code generators, which produce specific software from UML diagrams or other description of data [7]. Code generator tools cover areas that are less common than database handling or widgets layout. They are not, however, a general solution of the stereotyped code problem. There is no general approach behind these tools, and so they cannot be reused to produce code for other patterns.

Domain-specific languages (DSL, [14]) are an attempt to apply code generation to even narrower areas. In the DSL approach, a new small language is created for every specific code pattern. The language is applicable only in its domain: it is not general purpose language, and may even not be Turing-complete. DSL are widely used in program engineering.

The obvious way to create DSL is to define its grammar, then write or generate a parser and implement a translation scheme on certain program language. Language analysis requires substantial additional competence of a programmer and therefore sets an “entry threshold“ that limits the code generation availability. In many cases, it is simplier just to write a templated code manually than to design a new language. There also exist several tools for DSL creation, the most prominent being Visual Studio DSL Tools [4] or JetBrains Meta-Programming system [6]. These tools have immense capabilities and allow creation of very complex models and languages. However, studying these tools is even more complex than writing DSL compiler manually.

Our concern is to decrease a complexity of DSL creation and therefore improve a code generation availability. We want to make code generation so simple that its usage would be reasonable even in simpliest cases. We explore a different approach to domain-specific languages. By our observations, domain-specific *language* is not required in many cases. We may use the same general-purpose markup language (like XML) in different domains to describe a desired program code. However, domain-specific *semantics* is still needed, because the way the description is to be processed changes with each domain. We propose a new way to create code generators: a language with separted semantics. The language only defines how we should markup the data with commands, and does not specify what the commands mean. The logic of commands is written in arbitrary programming language for each domain.

Our approach to code generators resembles the Common Gate Interface (CGI) approach to web applications. CGI does not require invention of a special language for each web site, like the DSL approach. It does not make us write

individual programs that analyze HTTP packages. Instead, CGI parses packages itself and adopts environment variables and standard input to carry information from the package. Then CGI launches an arbitrary program, which processes the request, accepts an output of the program and sends it back to the user over the network. In spite of thorough research, we have not found this model to be implemented in any known programming language or tool.

Our approach to code generation was implemented in Thorn — a new programming language with separated semantics, designed especially for code generation. It is an Open source product under GPL v3 licence. Thorn is a fully operable software product, tested for more than four years, with successful practice in code generation.

In the first section, we describe Thorn language: the syntax of Thorn document, the way to create a new command and details of the way commands are executed. In section 2 we describe a relatively simple code generator for HTML and WordprocessingML [1] documents. In section 3 we describe the Thorn programming technique for creation of complex generators – functional generators. We also describe two examples of functional generators. The first generator converts bibliographic data from Thorn to BibTeX, HTML, WordprocessingML and some other formats. The second generator, described in section 4, is rather a programming framework that consists of Thorn libraries and C# auxiliary assemblies.

II. Thorn essentials

Thorn document is a tree of nested commands. The example of Thorn syntax is shown in Listing 1. We may compare the Thorn document with the corresponding HTML document which is shown in Listing 2.

Listing 1 Thorn code for a table.

```
\table[border=1 align=center] {
  \tr {
    \td {a11} \td {a12}
  }
  \tr {
    \td {a21} \td {a22}
  }
}
```

Listing 2 HTML code for a table.

```
<table border=1 align=center>
  <tr>
    <td> a11 </td> <td> a12 </td>
  </tr><tr>
    <td> a21 </td> <td> a22 </td>
  </tr>
</table>
```

Thorn document is more compact than HTML. Its size may be further decreased. Firstly, if the order of parameters of a command is fixed, their names may be omitted: `\table[1 center]`. Secondly, we may define the nesting of commands. We know `\tr` command is always inside `\table`, and `\td` is inside `\tr`. Therefore, curly brackets may also be omitted. The resulting code is shown in Listing 3.

Listing 3 Compact Thorn code for a table.

```
\table[1 center]
  \tr \td a11 \td a12
  \tr \td a21 \td a22
```

Parsing of input stream is provided by pushdown automaton. It should be mentioned that due to curly brackets ommiting, Thorn grammar is ambiguous. Depending on commands' definitions, the string `\cmd1 \cmd2` may be interpreted as `\cmd1{\cmd2{}}` or `\cmd1{}\cmd2{}`, which implies different parse trees.

The order of parameters' names, the rules of nesting and the logic of commands are defined in command files. The set of such files (library) must be loaded by Thorn before document compilation starts. The example of `\table` command file is shown in Listing 4.

Listing 4 Declaration of `\table` command.

```
#Keys=border,align;Blocks=entry;
#Parents=body;Type=Perl;Free=yes;

$STRING="<table border=$PARAM{border}
                 align=$PARAM{align}>
         $TEXT{entry}
         </table>";
```

The command description starts with service section, which is marked with # symbol (lines 1-2 in Listing 4). The service section specifies the default order of two parameters, `border` and `align`. Their names may be omitted, as in Listing 3. The command description also specifies the name of text entry to use inside the program. Other parameters specify the command, which can contain `\table`, the language the command logic is programmed in, the fact that we may omit curly brackets for `table` command.

After the service section is completed, the program in Perl is written. This program fills `$STRING` variable, which is an output of the program. It uses special hashes `%PARAM` and `%TEXT`, which store text variables from the input document. Aside from special variables, the program is arbitrary: it can manipulate files, use modules, etc. The program is executed not by Thorn, but by the Perl compiler. When Thorn acquires all the information about the command `\table` (in Listings 1 and 3, these are values '1' and '`center`' and the result of execution of two `tr` commands), it launches the Perl compiler with 'processor' program, and stores information about command and its parameters to STDIN. The processor reads the information from STDIN, executes the command code by `eval` function, and prints the result to STDOUT.

Thorn compiler reads STDOUT, removes the command from the document and places the result in corresponding place.

Commands may return an error message, which will be passed to the user. Commands may interact by global variables, stored in `%GLOBAL` hash. Processor obtains `%GLOBAL` hash from Thorn compiler and sends it back each time. Macros are also available: the result of a macro command will be processed by Thorn again.

Currently, there are three ways to execute the Thorn command:

- Use Perl commands;
- Make the new type of command and write a plugin that executes this type. This includes addition of new script languages in Thorn;
- Reference Thorn.dll in a .NET project, associate a command name with a class that implements corresponded interface, and launch Thorn compiler from the project.

Commands can be documented in Thorn language. A comment section is placed before a command section with ## marker. It contains a commentary in Thorn language with special commands like `\desc` (general description), `\key[i]` or `block[i]` (i-th parameter or text block description), etc.

Let us give an example of basic code generation of C# code. Consider a command `\event` in Listing 5. This command

Listing 5 Command for generation of events.

```
#Type=Perl; Keys=Type,Name; Blocks=Comment;
$type=$PARAM{Type}EventHandler;
$name=$PARAM{Name}EventSize;
$args=$PARAM{Type}EventArgs;

$STRING.="
///<summary>
///$TEXT{Comment}
///</summary>
public event $type $name;

///<summary>Raises $name</summary>
protected virtual void On$name($args e) {
   if ($name!=null) $name(this,e);
}";
```

will transform the following Thorn code:

```
\event[Mouse MouseMove]{Comment}
```

into a declaration of event and corresponding invocation method.

Several simple libraries for generation of C# code have been developed. Commands in these libraries can generate properties (with custom access modifiers and optional invocation of event), events, enumerations, `switch` operators (in case they are large and nested) and other templates for fast C# programming. We can use commands from different libraries in the same document. It corresponds to merging code generator applications, but does not require any special efforts.

Comparison to XML and XSL approach. In certain degree, Thorn follows XML/XSLT approach to HTML/CSS generation. In this approach, data is written in an XML document. The document is then converted into HTML with XSLT. It is even possible to use XSL to convert XML document into Java source code [3].

We argue that Thorn is more comfortable for code generation than XML/XSLT technology. The key difference is a possibility to use an arbitrary language for commands' logic. That simplifies generators greatly. Note how natural and readable listing 5 looks, especially in comparison with XSLT schemata in [3]. We may develop different commands in different languages, therefore choosing the most fitting language. In addition, we may create new command types and therefore patterns for commands' logic, as it will be shown below.

It is possible to develop an XML compiler that acts exactly as Thorn compiler: parses tags and processes them with Perl. The reason why we have developed a new language is that we wanted to minimize manual typing and therefore make Thorn more comfortable for code generation. Still, Thorn approach to compilation can be applied to other languages.

Comparison to DSL approach. Thorn language itself is not domain-specific, since its semantics is not defined. However, Thorn with selected set of libraries can be considered as a domain-specific language. Therefore, Thorn can be viewed as a tool for DSL creation. We believe that Thorn is much simplier than other such tools. There is no need to describe tokens, write down language grammar, etc. To create language semantics we only need to write simple Perl programs, and demands for these programs to be Thorn commands are not burdensome. Summarizing, Thorn has a very low "entry threshold" and can be used for fast creation of small DSL for a project, hence making code generation more available.

Thorn can be also used as a back-end for a compiler. Commands of Thorn form a tree, therefore a parse tree of a front-end compiler can be stored as a Thorn document and then interpreted with appropriate library.

III. Code generation for markup languages

In this section, we consider generation of documents written in markup languages [11]. We have developed a library for producing an HTML code from Thorn description. It supports all HTML tags. It defines nesting of tags so curly brackets are rarely used. It specifies orders of most popular parameters of tags. In addition, if a parameter name starts with `!` symbol (`!SomeStyle`), it will be placed in `style` attribute as `SomeStyle=Something`. All these improvements make Thorn files very small and readable in comparison with generated HTML.

All commands for HTML tags perform the same logic. The name of a command is translated into a tag, all attributes are listed after this tag with their names, the only text block is placed inside a tag, etc. Hence, we actually do not need to program the command logic in Perl, as in Listing 4. Instead, we develop new command types. In command declaration,

we write `Type=HTMLPairedTag` instead of `Type=Perl`. Commands of this type are executed by the Thorn compiler itself, without launching the Perl compiler. It improves performance greatly, since the major time of Thorn work lies in passing parameters between the compilers.

The second library is a library for producing WordprocessingML [1]. WordprocessingML is an XML dialect for text processors. This standard is supported by Microsoft Word, Open Office Word Processor and other text processors. Output in WordprocessingML format allows using all features of text processors. However, WordprocessingML files are not easy to type. The first reason is XML being redundant. The second reason is that WordprocessingML reflects the logic of a word processor, but not of a human. For example, items of multi-level lists are not really nested within each other, as in HTML or TeX. Instead, each item is a paragraph, and its level is determined by a style. Bold and italic words are not embedded in a plain text. Instead, the plain text ends, then a text with bold style starts and ends, and then plain text continues. There are many other similar inconveniences. We have developed the Thorn library, which allows to write Thorn documents in a habitual way (very much like TEX) and then to transform them into WordprocessingML. Not all features of WordprocessingML are currently supported.

A special extension for both these libraries (and potentially for any library that produces text documents) is created. This extension allows to create not only a document, but also a program that produces this document. Consider the code in Listing 6.

Listing 6 A document with a variable inside.

```
\document \html \body

\p Variable equals
   \variable[name=Var type=Int default=5]
```

Commands `\document` and `\variable` are defined in two libraries: lib.programmer and lib.makerup. In lib.makerup, `\document` does nothing, it only returns its entry. The command `\variable` returns default value ('5' in Listing 6). Therefore, the maker-up sees an example of a document, which will be created by a program. In lib.programmer, these commands are defined differently. Command `\variable` is transformed into a marker, which separates the text into variable and invariable parts. Command `\document` assembles parts and produces methods, which take all variables as arguments, and store a resulting document in a stream. There are two versions of lib.programmer: PHP version for web sites and C# version for offline software.

IV. Functional generators

When a code generator becomes more complicated, parameters of each command become numerous and hard to remember. We need to divide one command into several. This can be done by using relations. The relation is a set of records. In each record, some fields are specified. Instead of producing output, Thorn commands fill a relation in global variables. In many cases, a generator that translates a relation description into source code can be represented as a following function (which we call functional generator):

$$\begin{array}{rlllll} p(A) = h(& g_1[& f_1(A_1), & \ldots, & f_1(A_m) &], \\ & g_2[& f_2(A_1), & \ldots, & f_2(A_m) &], \\ & \ldots & & & & \\ & g_n[& f_n(A_1), & \ldots, & f_n(A_m) &]). \end{array}$$

Here h produces the source code. In an object-oriented program, h typically produces one class. Functions $g_1, \ldots, g_n$ produce parts of code, for example, methods inside the class. Functions $f_1, \ldots, f_n$ produce parts of methods, corresponding to one record in the relation. $A_1, \ldots, A_m$ are records of the relation. Usually, g_i are concatenation functions, i.e. $g_i(x_1, \ldots, x_n) = x_1 \cdot \ldots \cdot x_n$, where $\cdot$ denotes the concatenation operation. Also, h can usually be represented as

$$h(x_1, x_2, \ldots, x_n) = a_0 \cdot x_1 \cdot a_1 \cdot x_2 \cdot a_3 \cdot \ldots \cdot a_n \cdot x_n \cdot a_{n+1},$$

where a_i are string constants.

A simple way of representing relations in global variables is chosen. A value of field `Field` in a record with number `N` in a relation `Relation` is stored in global variable `Relation#N#Field`. This representation is supported by a new type of Thorn commands, as with HTML commands. We have also developed a Perl module that provides user-friendly way to access relation in global variables. A set of methods to implement functional generators in Thorn and Perl is called Fungi (functional generator interface).

Based on Fungi, we have developed BibThorn, an analogy to BibTeX. BibTeX is a flexible and widespread technology for storing bibliographic data. However, TeX can generate only a small set of formats. TeX cannot produce HTML files to place bibliography on a web site, or a plain text to include in a scientific report. Using Fungi, we may describe bibliography as a relation. A simplified example is placed in Listing 7.

Listing 7 A bibliography information on Thorn.

```
\bibliography
    \item[book]
         \author John Smith
         \title My book
    ...
\print
```

Command `\item` adds a new record in a relation. Commands `\author` and `\title` fill corresponding fields. The logic of `\author` command could be encoded on Perl as in Listing 8.

Module `db.pm` is a Fungi implementation for Perl. Is allows easy access to relations in global variables. Since such commands are required for authors, title, publisher and other fields, their logic is stereotyped. Command `\author` may therefore be described as in Listing 9.

Listing 8 Perl implementation of \author

```
#Blocks=Author;Free=yes;
#Parents=item;Type=Perl;
require 'db.pm';
$db=db->new(\%GLOBAL);
$db->SetFieldInCurrentRow
     ("Authors",$TEXT{Author});
```

Listing 9 Fungi implementation of \author

```
#Blocks=Author;Free=yes;
#Parents=item;Type=FungiSetter;
```

Command \print actually prints the bibliography. The prototype of the \print command is shown in Listing 10.

Listing 10 Using Fungi in Perl command to create a bibliography.

```
#Type=Perl;
require 'db.pm';
sub MakeItem {
    %h=@_;
    $STRING.="$h{Author}. $h{Name}\n";
    }
$db=db->new(\%GLOBAL);
$db->RunOver(\&MakeItem);
```

RunOver method looks through records, copies each record into a hash, launches MakeItem method and passes the hash to the method.

V. Thornado framework

Thornado is a framework that allows fast creation of business software [12]. The main aim of Thornado is assistance in input and output of data. It contains a code generator for data description, and a .NET library with useful templates. Listing 11 demonstrates a description of one field.

Listing 11 Description of one field in the extended system for data description.

```
\field[string Email]
 \desc E-mail address
 \io TypeIO.String
 \gui Text
 \check
  \err.W v==""
     -- Address must be entered
  \err.W v.Length<5
     -- Address is too short
  if (!v.Contains('@'))
     list.Add("Address is not valid");
```

Command \field specifies the type of the field and its name. Command \desc specifies the commentary for this field in the generated code. It also specifies the caption for this field in graphic user interfaces.

Command \io specifies an object that transforms a field into a string and parses it from string. In C#, primitive types (int, double, etc.) can be written and parsed from a string by .NET means. Unfortunately, many types do not support parsing and writing in human-readable forms. The way to specify how exactly the object should be converted is often entangled and inconvinient. There is also no way to read and write null value. Therefore, we introduce TypeIO classes for input and output objects. For example, methods TypeIO.Int.Write and TypeIO.Int.Parse convert int value into a string and vice versa. TypeIO.Int is a predefined object of IntIO class. Many formats for the same type may exist: for example, double can be processed with TypeIO.Double, TypeIO.Double.Money or TypeIO.Double.Percent. Many TypeIO classes are developed, including those for classes that are not usually converted to string (like colors, pens and brushes). In addition, each TypeIO object X has properties X.Nullable that may process null and X.InArray that may process arrays.

Command \gui specifies a type of graphic user interface. Several types are available. TextBox option allows input of an arbitrary string with following conversion into a value with specified TypeIO object. ListBox means selecting one value from a list. In this case, \values command allows to specify items of the list, which is arbitrary IEnumarable object. Note that it is impossible to specify an object as an attribute in C#.

Finally, \check command describes the business logic of this field. The logic can be described by Thorn commands (\err, for example). It is also possible to place pure C# code into \check command with predefined variables v (a new value of the field) and list (a list that stores errors). This code will be placed in corresponding location of a generated source code. Business logic for one field is to be inserted into corresponding property. Business logic for the whole class is to be placed into a method CheckConsistancy. The method is called before input-output procedure begins and after it has been completed.

Thornado can generate a class with required fields and business logic. Each field of generated class is associated with FieldInfo object. FieldInfo contains all the information we specify in Thorn file: caption for a field, a type of graphic user interface, etc. Thorn can generate IOProvider, which stores a collection of FieldInfo objects and can manipulate fields of generated class. Other classes perform various operations by using IOProvider: input and output into INI- and XML-files, interconnection with ADO.NET, generating of graphic user interface widgets, etc. Therefore, we can generate a substantial part of an application from Thorn description.

IOProvider performs some sort of reflection. It is more convenient than traditional C# reflection. Names of

`FieldInfo` objects are C# variables and are checked in compile time, unlike string values that are used in reflection. `FieldInfo` objects carry all additional information in their fields, and there is no need to read attributes to access them. Finally, it is impossible to write in attributes neither the arbitrary business logic nor the references to other classes and objects.

VI. Conclusion and Future works

In this paper, we presented Thorn language, studied its differences from other approaches to DSL creation and described its basic programming techniques. Thorn is a fully operable software product, available under GPL v3 licence. Its area of usage is creation of small code generators for simple code patterns. Due to its simplicity, Thorn seems to be more preferable than other tools for DSL creation, or manual writting DSL compilers.

We consider following ways to improve our product:

- *nix version of Thorn
- Plug-ins for Microsoft Visual Studio, Eclipse and other popular integrated development environments
- More languages will be supported ”from the box” for command development
- BibThorn will be extended and integrated with LaTeX editors
- Thornado framework will be further developed

References

[1] Standard ecma-376. http://www.ecma-international.org/publications/standards/Ecma-376.htm.

[2] Windows forms designer. http://msdn.microsoft.com/en-us/library/e06hs424(VS.80).aspx.

[3] E. M. Burke. *Java and XSLT*. O'Reilly, 2001.

[4] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[6] S. Dmitriev. Language oriented programming: The next programming paradigm. http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf, 2004.

[7] K. Fertalj and M. Brcic. A source code generator based on uml specification. *International journal on computers and communications*, 2(1), 2008.

[8] J. Harrington. *Code Generation in Action*. Manning, 2003.

[9] J. Levine. *Flex and Bison. Text processing tools*. O'Reilly Media, 2009.

[10] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997.

[11] Y. Okulovsky and D. Deyev. System of generation of documents in html, mht and wordprocessingml formats (russian). *Bulletin of Saint Petersburg State University of Information Technologies, Mechanics and Optics: Mechatronics, Technologies and Computer-aided design*, (70), 2010.

[12] Y. Okulovsky, D. Deyev, V. Popov, and V. Chasovskikh. Code-generation system thornado and its application to creation of business-software (russian). *Bulletin of Saint Petersburg State University of Information Technologies, Mechanics and Optics*, (87), 2008.

[13] A. Troelsen. *Pro C# 2010 and the .NET 4 Platform*. APress, 2010.

[14] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.