Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 263–271
doi: 10.14794/ICAI.9.2014.2.263

Domain-specific languages with custom operators*

Áron Baráth, Zoltán Porkoláb

Department of Programming Languages and Compilers,
Faculty of Informatics, Eötvös Loránd University
{baratharon,gsd}@caesar.elte.hu

Abstract

Embedded domain-specific languages have become more popular due to their expressive power. The tasks are composed at the level of the problem domain. This paper shows the key elements and the benefits of implementing embedded domain-specific languages in strongly typed imperative programming languages. These ideas are useful for those people who use or implement embedded domain-specific languages but they are not professionals in functional programming and prefer imperative programming languages instead.

The most common host language of the domain-specific languages is a functional programming language, because of the freedom to introduce custom operators. The investigation is based on a strongly typed imperative language which allows to define such operators. Domain-specific languages increase abstraction level and hide details. This paper explores other aspects of domain-specific languages for functional programmers.

The key language element is in functional languages to define custom operators. These language elements must be found in the host language for implementing an efficient embedded domain-specific language. In this paper there is an example language to demonstrate the expressive power. The investigation concludes that the benefits of using an imperative language to implement an embedded domain-specific language are the increased expressiveness by eliminating redundant parentheses' and unnecessary verbose function names from the source code. Based on the strong static type system the custom types preserve the type-correctness.

This paper shows ideas and recommendations about how to implement a domain-specific language in a strongly typed imperative language.

Keywords: custom operator, domain-specific language, static type-system, imperative language

MSC: 68N15

*The project was supported by Ericsson Hungary.

1. Introduction

Domain-specific languages (DSL) have become more popular due to their expressive power. These languages are designed to special purpose with specialized features. DSLs can be subdivided to markup languanges, modelling languages and *programming* languages. Some DSL kinds can be used for common domains, such as HTML, and for more specific purposes. DSLs are operating at higher level hiding details from the DSL user. The syntax varies in a wide range in order to be familiar to the experts of the domain.

Implementing a brand new DSL is very expensive since designing a new lexical analyser, and building a new parser have high costs. Selecting the execution context is a quite difficult task. Many DSLs are embedded into an existing language in order to reduce the costs. This language is called the *host language*. Therefore, a DSL uses the lexical analyser, the parser, and the semantic checker of the host language, which is comfortable. The embedding procedure extends the set of language elements of the host language. The new elements are implementing the special syntax of a DSL.

There are two different ways to embed a DSL into a host language. The *shallow embedding* provides the most flexible DSL syntax due to the fact that they are storing expressions in string form. This style has a disadvantage at runtime, because DSLs must parse its code at runtime and have to check a lot of conditions. Furthermore, the shallow embedding does not facilitate the compiler of the host language to check the written code. Checking the syntax and the semantic rules of the embedded code can be done with an external tool only.

The *deep embedding* solves the most shortages of the shallow embedding by being the extension of the host language. However, the deep embedding has more requirements than the shallow one: it needs an extendable syntax and a strong static type-system. The semantic rules can be defined with the type-system of the host language. The flexibility of the syntax of the deep embedded DSL depend on the host language. This dependency can be a disadvantage if the host language is not flexible enough. Most of the functional programming languages are quite flexible to meet the requirements. Embedded DSLs are usually developed in the Haskell programming language [1, 2], because Haskell is platform independent and it has a rich set of language elements. In this paper we consider only the deep embedded DSLs.

The special syntax of a DSL can be more familiar to its domain expert. Due to the special syntax, the domain expert is able to catch errors specific to the problem domain. DSLs often install new constructs to describe domain problems, or they even apply different programming paradigms. Thus, the syntax of a DSL may reflect the usual notations of the domain to make it usable for the domain experts. The capabilities of the DSL depend on the host langauge – hence, selecting the right host language is a difficult task. There are additional requirements to the host language. Several questions must be answered, such as: how to describe the DSL (declarative or imperative), where to use the DSL (piece of software or

stand-alone), who will use the DSL?

Although DSLs are indispensable in their domain, the DSL programs execute most of their actions out of that domain. The program will works with classical operations: creating threads, allocating low-level memory, opening network connections, and communicating with the operating system. These operations must done, but the user of the DSL do not have to know about this.

Nowadays many embedded domain-specific languages (eDSL) [3, 4, 5, 6, 7] are written in Haskell. The key language elements are the algebraic data type [8], and the operator overloading mechanism using custom operators. Algebraic data types make embedding simple, while the pattern matching grants type-safe processing by design.

The custom operators based on the strong type-system help the user of the eDSL to write understandable codes by eliminating long function names. The operators have symbols, precedence and associativity rules. Every well-adjusted property is beneficial; a meaningful operator symbol helps to document the source code, while good precedences and associativity rules can eliminate unneccessary and disturbing parentheses. Furthermore, using operators is beneficial to the DSL developer to introduce new operators easily.

The functional programming languages allow to introduce new operators, but the imperative languages rarely support this feature. In this paper, we introduce an imperative programming language called *ScriptKernel* (lately *Welltype*), which allows the usage of custom operators and provides a strong static type system. The language itself can be used as a DSL in many cases by extending new types and operators. The purpose of our language is to perform massive compile-time checks, and generate a fast binary which can be executed on various operating systems and architectures. The compiled program (binary) can ran on a virtual machine, providing portability and binary compatibility.

This paper is organized as follows: In Section 2 we describe the implementation details of custom operators in ScriptKernel. We show the used techniques on a complex number DSL example, which DSL is embedded into the ScriptKernel. With this DSL we can define custom operators. We evaluate the custom operator usage by an example in Section 3. The embedding type of the DSL affects the custom operator usage and expansion. Future plans are discussed in Section 4 such as solving problems by introducing algebraic data types in imperative languages. Our paper concludes in Section 5.

2. Implementation

Our programming language – *ScriptKernel*[1] – is not an interpreted, but a compiled imperative programming language. We are integrating into this language new functionalities and language elements taken from other paradigms. The target of the compilation is a custom virtual machine which supports multiple operating

[1] lately *Welltype*

system versions. The ScriptKernel is a programming language originally used in the computer game industry [9] but now has many advanced features. The language design is influenced mainly by C++ [10, 11], Ada [12], Eiffel [13], but some idea came from Haskell [2], C# [14] and Java [15].

The usage of custom operators brings the flexible type of the language: the *private record*. Many eDSLs use custom types to store private information. It is important to keep the internal state away from any accidental access. The private record is just a type name in the user program, and there is no information about the structure of the actual record. Other application of the private records are possible in the creation of new types especially composite types, such as complex numbers and 128 bit integer represented by two 64 bit integers. In both cases there are convenience reasons for implementing standard operators (`+`, `-`, `*`, etc). However, there are special functions using the complex numbers, for example the conjugate. We can overload an existing operator as follows:

```
function "~"(Complex c) : Complex
{
  Complex res;
  res.re, res.im = c.re, -res.im;
  return res;
}
```

Note that in this example we assume that the defintion of the function and the definition of the `Complex` type are placed in the same program. Otherwise, in a general case the type invariant can be violated without any notification, which is important when the program exports `Complex` as a private record. In this case we should export a *getter* and a *setter* for the real and the imaginary part as well.

```
export
{
  record Complex = private;
}
declare
{
  record Complex
  {
    double re;
    double im;
  }
}
function Re(Complex c) : double
{
  return c.re;
}
function Im(Complex c) : double
```

```
{
  return c.im;
}
```

But the `Complex` type does not have any invariants, so exporting it as a regular record is acceptable. Fortunately, the custom operator mechanism does not depend on the record type.

Before implementing a custom operator, the used operator symbol must be introduced. In the ScriptKernel it is considered the declaration of the operator class. This operator class consists of three components: the operator symbol, the associativity, and the precedence.

The symbol can be any sequence of `\`, `-`, `^`, `=`, `!`, `<`, `>`, `+`, `&`, `%`, `~`, `*`, `/`, excluding the `/*`, the `*/` and the `//`, because these are control sequences of the comments. The associativity must be `left` or `right`, and the precedence must be between 3 and 15 inclusive. A higher precedence implies that the operator will be evaluated faster than an operator with a lower precedence.

Regarding the complex numbers, we can define a new comparison for the complex conjugates. By definition, two complex numbers are complex conjugates if the real parts are the same and the imaginary parts have equal magnitude but opposite signs. The following code declares the new operator class of `=~=`. The new operator will be left associative with the precedency level of 10, similar to the built-in operator `==`.

```
declare
{
  operator =~= left 10;
}
```

After that, we can use the `=~=` symbol like any other operator: creating new operator functions to implement the operator functionality, then using it in expressions. The following code implements the operator of complex conjugates.

```
function "=~="(Complex lhs, Complex rhs) : bool
{
  return lhs.re==rhs.re && -lhs.im==rhs.im;
}
```

The next snippet shows how to use this `=~=` operator. In the `if` statement the condition must be an expression with type `bool`. The result type of the `=~=` operator

meets this requirement.

```
main
{
  Complex a, b;
  if(a =~= b)
  {
    write("They are complex conjugates.\n");
  }
}
```

The language provides default initializations for every type including record and private record too. The default values will be as follows: the default value is `0.0` for a variable of type `double`, `""` for `string`, `false` for `bool`, and so on. In the code snippet above, the two `Complex` variables `a` and `b` will be defaulted to the complex zero (real and imaginary parts equals to 0). This implies that the condition of the `if` statement will be true, and the message appears on the program's standard output.

2.1. Implementation details

This experimental language uses *flex* for lexical analyser and *bisonc++* for parser. These tools are successors of the *lex* and *yacc* tools [16]. The task is to implement the custom operator support with these tools. The problem is that, these tools do not provide any assistance to implementing operators with custom precedency level. The solution is to define the regular expression for the general operator symbols, and to set the token's value with the information of the defined operators. If the compiler context contains an operator class like `operator =~= left 10`, then the lexical analyzer knows this, and it returns the `OP_LEFT10` token when the `=~=` accepted. The related *flex* code is as follows:

```
...
{operator}          { return Parser::Lexer_Operator(this); }
...
```

The `Lexer_Operator()` function uses an `std::map` to look the operator symbol up, and if it is found, then it returns with the right token. If the given symbol is not found, the `UNBOUNDOPERATOR` token is returned to indicate the fact that the operator is unbound.

The `UNBOUNDOPERATOR` token is used in the parser's operator class declaring rule. The related *bisonc++* code is below. The operator symbol will be granted via a string. The associativity part is a common identifier. This is done to avoid the `left` and the `right` identifiers being keywords. The compiler will check the content of the identifier and report an error if it is neither the `left` or the `right`.

The range of the precedence level must be checked too.

```
...
  KW_OPERATOR UNBOUNDOPERATOR IDENTIFIER I32DEC SEMICOLON
  {
    ...
  }
...
```

If the paramteres are correct and the operator can be declared, then the operator will be inserted into the earlier mentioned map with the resultant token. In the *bisonc++* expression rules are instances for all possible operator tokens, so the precedences are still controlled by the *bisonc++*.

3. Evaluation

Modes of embedding a domain-specific language determines its flexibility and safety. In Haskell language the common embedding strategies are shallow and deep embedding. Shallow embedding is more flexible, but with high risk of errors and syntax errors even. Also deep embedding has some troubles with extending. The type of embedding cause the problem. In our language we show that the language can be used as the DSL, and there is not necessary to embedding literally.

Using operators within a source code can help to understand itself, because the reduced number of explicit function calls provides less distractions in the code. The importance of the custom operators is outstanding, therefore our language supports this.

An other perspective is the hot program updating. Because we have a virtual machine which can be used via library and as standalone tool, the loaded code can be replaced any time without stopping the host program. This feature is most commonly required, and native programs do not support this feature. Moreover the operating system resources can be supervised, because the virtual machine uses the functions of host program. This feature helps to protect the operating system from external attacks, so this is an important security reason to use a virtual machine. One of the famous language which supports hot code swap is the Erlang [17, 18].

4. Future works

We plan to develop the algebraic data types into our imperative language. Algebraic data types are very useful, for example we able to implement the union type with it, and this is a nice way to create recursive data types. The question is how to prepare the algebraic data type for an imperative programming language? There are some contradictions with this: in functional languages the pattern matching is a natural way to fork on the parameters. However, there is no usual way to match patterns in imperative languages.

When the algebraic data types are ready to use, the combination with the custom operators will open new freedom and opportunities in developing DSL languages. The combination of the flexibility and the safety will characterize the new version of our language.

5. Conclusion

In this paper we discussed how the expressive power of embedded Domain-specific languages can be increased by using custom operators. Instead of using verbose function names and disturbing parenteses clever operator overloading make the source code more clean and understable. In this paper we introduced an imperative programming language – ScriptKernel – which allows custom operators and provides a strong static type system. The language itself can be used as the DSL in many cases with extending new types and operators.

We described the overloading mechanizm and its implementational details of ScriptKernel. We used an example to evaluate the steps and the technical background of the custom operator mechanism in our language. The virtual machine guarantees the synchronization of the operator classes and the operator functions, so using custom operators never get contradictions.

References

[1] Hutton, G.: Programming in Haskell. Cambridge University Press (2007)

[2] O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell. O'Reilly Media (2008)

[3] Fowler, M.: Domain-specific languages. Addison-Wesley Professional (2010)

[4] Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages. Centrum voor Wiskunde en Informatika (2000)

[5] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM computing surveys (CSUR) **37**(4) (2005) 316–344

[6] Hudak, P.: Building domain-specific embedded languages. ACM computing surveys **28**(4es) (1996) 196

[7] Wikipedia: Domain-specific language (2013) http://en.wikipedia.org/wiki/Domain-specific_language.

[8] Jay, B.: Algebraic data types. In: Pattern Calculus. Springer (2009) 149–160

[9] Baráth, Á.: ScriptKernel (2013) https://people.inf.elte.hu/baaqaai/skc.

[10] Stroustrup, B.: The Design and Evolution of C++. 1st edn. Addison-Wesley (1994)

[11] Stroustrup, B.: The C++ Programming Language. 4th edn. Addison-Wesley Professional (2013)

[12] Barnes, J.: Programming in Ada 95. 2nd edn. Addison-Wesley (1998)

[13] Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice Hall (1997)

[14] Albahari, J., Albahari, B.: C# 4.0 in a Nutshell: The Definitive Reference. O'Reilly Media (2010)

[15] Bloch, J.: Effective Java. 2nd edn. Addison-Wesley (2008)

[16] Mason, T., Brown, D., et al.: Lex & yacc. O'Reilly Media, Inc. (1992)

[17] Laurent, S.S.: Introducing Erlang. O'Reilly Media (2013)

[18] J., A., M., W., C., W., R., V.: Concurrent Programming in Erlang. Prentice-Hall (1996)