Building Domain Specific Languages with the Metacasanova meta-compiler

Francesco Di Giacomo

Università Ca' Foscari di Venezia - PhD in Computer Science

- They allow to express the solution of a problem in a more natural way.
- They provide constructs that are domain-specific not provided by GPL's.
- They allow to develop complete application programs for a specific domain more quickly.

- They allow to express the solution of a problem in a more natural way.
- They provide constructs that are domain-specific not provided by GPL's.
- They allow to develop complete application programs for a specific domain more quickly.

CONSEQUENCE: It is desirable to deploy a DSL's when the scope of the application is very specific.

DSL	Application
Lex and Yacc	program lexing and parsing
PERL	text/file manipulation/scripting
VHDL	hardware description
TEX, LATEX, troff	document layout
HTML, SGML	document markup
SQL, LDL, QUEL	databases
pic, postscript	2D graphics
Open GL	high-level 3D graphics
Tcl, Tk	GUI scripting
Mathematica, Maple	symbolic computation
AutoLisp/AutoCAD	computer aided design
Csh	OS scripting (Unix)
IDL	component technology (COM/CORBA)
Emacs Lisp	text editing
Prolog	logic
Visual Basic	scripting and more
Excel Macro Language	spreadsheets and many things never intended

PROBLEM: Creating DSL's requires to implement a compiler/interpreter for the language.

- Compilers are complex
- Several modules: parser, type checker, code generator/code interpreter
- Require a lot of development time.
- Not flexible: adding features to the language compiled by hard-coded compilers takes a considerable effort.

The implementation of the compiler is a repetitive process:

- The parser can be created with parser generators (e.g. Yacc)
- The type system must be implemented in the host language.
- The operational semantics must be reflected in the generated code (code generations).

How they are formalized:

- Expressed in a form that mimics logical rules.
- They are compact.
- They are readable.

How they are implemented:

- Encoded with the abstractions provided by the host language.
- Readability is usually lost in the translation process.
- The effort required for the translation is high.

Semantics of a statement that waits ford a condition or a certain amount of seconds:

$$\frac{\langle t - dt > 0 \rangle \Rightarrow \texttt{true}}{\langle \texttt{wait}\ t; k\ dt \rangle \Rightarrow \langle \texttt{wait}\ t - dt; k\ dt \rangle}$$

$$\frac{\langle t - dt > 0 \rangle \Rightarrow \texttt{false}}{\langle \texttt{wait}\ t; k\ dt \rangle \Rightarrow \langle k\ dt \rangle}$$

$$\frac{\langle c \rangle \Rightarrow \texttt{true}}{\langle \texttt{wait}\ c; k\ dt \rangle \Rightarrow \langle k\ dt \rangle}$$

$$\frac{\langle c \rangle \Rightarrow \texttt{false}}{\langle \texttt{wait}\ c; k\ dt \rangle \Rightarrow \langle \texttt{wait}\ c; k\ dt \rangle}$$

STUB Paste the code for wait state machine from Casanova compiler

- Formal semantics provide a clear, compact, and simple way to describe the constructs of the language.
- Implementing a compiler in a GPL requires to encode the semantics within the abstractions provided by it.
- The result is something completely different.

Why not implementing a program that can take as input the language definition expressed in the fashion of the semantics rules, a program written in that language, and outputs executable code?

Why not implementing a program that can take as input the language definition expressed in the fashion of the semantics rules, a program written in that language, and outputs executable code?

We can: this software is defined as meta-compiler.

- Metacasanova is a metacompiler that uses semantics rules to define a language.
- A program in Meta-casanova contains:
  - Data declarations
  - Function declarations
  - Subtypes declarations
  - Rules

Below the code for:

- Integer constant
- Sum of arithmetic expressions
- While loop

```
Data "$i" -> <<int>> : Value Priority 300
Data Expr -> "+" -> Expr : Expr Priority 100
Data "while" -> Expr -> Expr : Expr Priority 10
```

Below the code for:

- Declaration of evaluation of expressions.
- Function to add a variable to the symbol table.

```
Func "eval" -> Expr : RuntimeOp =>
    EvaluationResult Priority 0
Func SymbolTable -> "defineVariable" -> Id :
    MemoryOp => SymbolTable Priority 300
```

In the code below we declare that a value, id, and a sequence of expressions are considered expressions as well.

```
Value is Expr
Id is Expr
ExprList is Expr
```

- A rule is made of a set of premises and a conclusion. A premise can be a function call or a predicate.
- First the left part of the conclusion is evaluated.
- The language does pattern matching on the function arguments and if it fails the rule is not executed.
- The premises are evaluated in order. The language looks for possible candidate rules for the execution, i.e. rules containing in the conclusion the function called by the premise. In case of predicates, the predicate is immediately evaluated.
- If the rule call fails, then the entire rule evaluation fails.

Consider:

- A set of rules R.
- A set of function calls F for each rule in R.
- A set of clauses C for each rule in R.
- The notation f^r means apply the function f through the rule r.
- The notation $\langle f^r \rangle$ means evaluating the application of f through r.
- The result of a rule is in general a set because it is possible to allow the rule to branch, i.e. if the evaluation of a premise succeeds by using more than one rule, we return all the possible result.

$$\text{R1: } \frac{\begin{array}{c} C = \emptyset \\ F = \emptyset \end{array}}{\langle f^r \rangle \Rightarrow x}$$

$$\text{R2: } \frac{\begin{array}{c} \forall c_i \in C \ , \langle c_i \rangle \Rightarrow true \\ \forall f_j \in F \ \exists r_k \in R \mid \langle f_j^{r_k} \rangle \Rightarrow \{x_{k_1}, x_{k_2}, ..., x_{k_m}\} \end{array}}{\langle f^r \rangle \Rightarrow \{x_1, x_2, ..., x_n\}}$$

$$\text{R3(A): } \frac{\exists c_i \in C \mid \langle c_i \rangle \Rightarrow false}{\langle f^r \rangle \Rightarrow \emptyset}$$

$$\text{R3(B) } \frac{\forall r_k \in R \ \exists f_j \in F \mid \langle f_j^{r_k} \rangle \Rightarrow \emptyset}{\langle f^r \rangle \Rightarrow \emptyset}$$

```
----------------------
eval ($i v) => ($i v)
```

- The rule above does not contain premises, i.e. it is an axiom.
- If the pattern matching on the conclusion succeeds, i.e. the input is an integer constant, then we simply return the constant as evaluation of the expression.

```
eval expr1 => ($i val1)
eval expr2 => ($i val2)
<<val1 + val2>> => arithmeticResult
-------------------------------
eval expr1 + expr2 => $i arithmeticResult
```

- The first rule does pattern matching on `expr1 + expr2`. This means that if the input is not in that form, the rule is not triggered.
- In the premises, if the result of the recursive call to the evaluation function is not an integer constant, the rule evaluation fails. This might happen when executing, for example, the sum of two strings.
- The third premises emits C# code to do the sum computation.
- The result is another integer constant containing the sum of the two expressions.