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:

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

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

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

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

Towrds meta-compilers
Implementation of the semantics

```
| t_expr, OptimizedQueryAST.Wait((tp, expr), _) ->
  let float_type = TypeDecl.ImportedType(typeof<float32>, t_expr.Position)
  let bool_type = TypeDecl.ImportedType(typeof<bool>, t_expr.Position)
  e.CountdownCounter <- e.CountdownCounter + 1
  let tp = TypeDecl.ImportedType(typeof<float32>, tp.Position)
  let lb, lb_expr = get_fresh_label t_expr.Position e
  let count_down_id = {idText = "count_down" + (string e.CountdownCounter); idRange = t_expr.Position}

  let f_expr =
    match expr with
    | f_expr when tp = TypedAST.TypeDecl.ImportedType(typeof<float32>, t_expr.Position) -> f_expr
//    | i_expr when tp = TypedAST.TypeDecl.ImportedType(typeof<int>, t_expr.Position) -> TypedAST.ImplicitIntCast(i_expr)
    | _ -> raise "Sate machines error. Wait arguments can be either of type int or float."

  let count_down =  TypeDecl.ImportedType(typeof<float32>,t_expr.Position), Expression.Var(count_down_id , tp, traverseTrivialTypedExpr (tp, f_expr) |> Some)
  let wait_lb, wait_lb_expr = get_fresh_label t_expr.Position e
  let goto_wait_lb = (TypeDecl.Unit t_expr.Position, Goto.Create wait_lb |> Expression.Goto)
  let _then =
   [(TypeDecl.Unit t_expr.Position,
     Expression.Set(count_down_id,
                    (float_type,
                     StateMachinesAST.Sub((float_type, StateMachinesAST.Id(count_down_id)),
                                          (float_type, StateMachinesAST.Id({idText = "dt"; idRange = t_expr.Position}))))));
    (TypeDecl.Unit t_expr.Position, GotoSuspend.Create wait_lb |> Expression.GotoSuspend)]
  let _else = [exit_expr]
  let cond = bool_type, Greater((float_type, Id(count_down_id)), (float_type,StateMachinesAST.Expression.Literal(BasicAST.Float(0.0f, t_expr.Position))))
  let if_then_else = TypeDecl.Unit tp.Position, Expression.IfThenElse(cond,None, _then, _else)
  lb, [lb_expr; count_down; goto_wait_lb; wait_lb_expr; if_then_else]

| t_expr, OptimizedQueryAST.ReEvaluateRule(p) ->

  let lb, lb_expr = get_fresh_label t_expr.Position e
  let goto_exit =
    match exit_expr with
    | t, GotoSuspend(g) -> t, Goto(Goto.Create(g.Label))
    | t, Goto(g) -> exit_expr
```

- 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.

Case study: Casanova 2.5

- **Entities:** They contain both the data and the behaviour of the objects in the game
- **Rules:** They define the behaviour of the entity. Once a rule ends its execution it is restarted at the next frame.
- **Domain:** A set of entity attributes the rule is allowed to change. A rule can always read all fields but can modify only those in the domain through a `yield` statement.

Example of program in Casanova

```
entity Guard = {
Position   : Vector2
Velocity   : Vector2

rule Position = Position + Velocity * dt

rule Velocity =
wait Position.X >= 300f || Position.X <= 0f
yield new Vector2(-Velocity.X, 0f)
}
```

Rule pausing

Rule execution can be paused with built-in statements:

- `wait` takes either a floating point value or a predicate. In the first version the rule is paused by the given amount of seconds, in the second it is paused until the condition is met.
- `yield` updates the fields in the domain with the given values. The rule execution is paused by one frame in order to be able to see the changes at the next game update.

Semantics of wait

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

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

Implementation of wait in Metacasanova

In the following waiting on a condition is called **when** because Metacasanova does not currently support operators overloading

```
eval expr ctxt => ($f t)
<<t <= dt>> == false
-----------------------------------
eval_s (wait expr) k ctxt dt => Suspend (wait $f (<<t - dt>>));k ctxt

eval expr ctxt => ($f t)
<<t <= dt>> == true
-----------------------------------
eval_s (wait expr) k ctxt dt => Resume k ctxt


eval expr ctxt => ($b true)
-------------------------------------------
eval_s (when expr) k ctxt dt => Atomic k ctxt

eval expr ctxt => ($b false)
-------------------------------------------
eval_s (when expr) k ctxt dt => Suspend (when expr);k ctxt
```

Results: patrol script

- Script making a guard patrol two checkpoints
- Tests run with the script written in Casanova 2.5 and Python.
- Implementation of Casanova 2.5 compared with the code length of the hard-coded compiler modules for Casanova 2.0.
- Python is used as a scripting language in several games (e.g. World in Conflict, Civilization IV).
- We neglected C# because we already have benchmarks for that in Casanova 2.0 and both languages are faster of several orders of magnitude.

Results: patrol script

Casanova 2.5		
Entity #	Average update time (ms)	Frame rate
100	0.00349	286.53
250	0.00911	109.77
500	0.01716	58.275
750	0.02597	38.506
1000	0.03527	28.353
Python		
Entity #	Average update time (ms)	Frame rate
100	0.00132	756.37
250	0.00342	292.05
500	0.00678	147.54
750	0.01087	91.988
1000	0.01408	71.002

Table: Patrol sample evaluation

Results: patrol script

Casanova 2.5 with Metacasanova	
Module	Code lines
Data structures and function definitions	40
Query Evaluation	16
While loop	4
For loop	5
If-then-else	4
When	4
Wait	6
Yield	10
Additional rules for Casanova program evaluation	40
Additional rules for basic expression evaluation	201
Total: 300	
Casanova 2.0 compiler	
Module	Code lines
While loop	10
For-loop and query evaluation	44
If-Then-Else	15
When	11
Wait	24
Yield	29
Additional structures for rule evaluation	63
Structures for state machine generations	754
Code generation	530
Total: 1480	

Table: Metacasanova vs Casanova 2.0 compiler

Conclusion

Advantages:

- The code length to define the semantics of the language is almost 5 times shorter.
- Using Metacasanova greatly reduces the effort to build a compiler.
- The definition of the language in Metacasanova is almost identical to the formal definition.
- Extending the language becomes easier.

Disadvantages:

- The generated code is slower than Python but of the same order of magnitude.
- Code performance affected by maps used to model the memory.
- The code generation requires further improvement.
- Using data structures and methods from external libraries requires to build a wrapper with meta-types and meta-functions.

Related work

- META II family languages, parser generators plus code generators. Untyped. (1970's).
- RML language family based on natural semantics (1996). It generates C code.

Future work

- Type classes and inliner to access fields in records by using the result of type function calls.
- Handling imported libraries automatically by generating `unsafe` type to handle side effects, or allow the programmer to specify a mapping to metacompiler types.
- Expand Casanova 2.5 with networking primitives.

Tank you!

Questions?