Metacasanova: An Optimized Meta-compiler for Domain-Specific Languages

Francesco Di Giacomo [1]
Mohamed Abbadi [3]
Agostino Cortesi [1]
Pieter Spronck [2]
Giuseppe Maggiore [3]

Università Ca' Foscari - Venice

Tilburg University - Tilburg

Hogeschool Rotterdam - Rotterdam

Summary

We present Metacasanova, a meta-compiler initially created to ease the development of Casanova, a DSL for game development.

Topics:

- Introduction on DSL.
- Introduction of Metacasanova and example of use.
- Language extension with Functors.
- Example of Records implemented with Functors.
- Results and Conclusion.

- Abstractions that are closer to the problem domain.
- Speed-up of development time of the problem solution.

Two possible paths:

1. Embed the DSL in a host language.
2. Write a compiler/interpreter.

Embedding Pros and Cons:

- Re-use of the host language infrastructure.
- Developers expert in the host-language need only to become familiar with the language extension.
- Syntax and type system bound to those of the host language. Type system as well.
- Domain-specific optimizations are difficult.

Compilation/interpretation Pros and Cons

- Syntax and types correspond to the language definition.
- Good error reporting.
- Domain-specific optimizations are possible.
- Long development time.
- Development process follows recurrent patterns.

1. Formalize the grammar of the language.
2. Formalize the type system and semantics.
3. Build a syntactical analyser.
4. Build a type checker.
5. Implement the semantics in the target language.

- Step 1 and 2 are creative and cannot be captured by a pattern.
- Step 3 can be completed by using a Lexer/Parser generator.
- Step 4 processes the result of the syntax analysis and implements the formalization of the type system in a chosen programming language.
- Step 5 takes the result of Step 4 and implements the formalization of the semantics in the target language.

Step 4 and 5 follow the same pattern independently of the language we are building the compiler for.

- The implementation mimics the behaviour of the meta-representation of the formal semantics.
- The formalization of the types and semantics is lost when implemented with the abstraction of the chosen programming language.
- Example: if we use inference rules, then we re-implement their behaviour in the host language each time we write a new compiler.

Goal: Express the repetitive steps in terms of the formalization.

Research questions

Research question 1: *To what extent does Metacasanova ease the development speed of a compiler for a Domain-Specific Language, in terms of code length compared to the hard-coded implementation, and how much does the abstraction layer of the Metacompiler affect the performance of the generated code?*

Research questions

Research question 1: *To what extent does Metacasanova ease the development speed of a compiler for a Domain-Specific Language, in terms of code length compared to the hard-coded implementation, and how much does the abstraction layer of the Metacompiler affect the performance of the generated code?*

Research question 2: *In what way can we embed the type system of the implemented language in Metacasanova in order to get rid of the dynamic lookups at runtime and what is the performance gain of this optimization?*

Metacompilers

- Input: the definition of a language in a meta-language.
- Input: a program written in that language.
- Output: Executable code for the program.

Metacasanova

- Input: Language definition in terms of inference rules.
- Input: A meta-representation of the program in that language.
- Output: C# code (it can later be compiled using a .NET compiler).

- Meta-data structure declarations: used to represent the abstractions of the language.
- Function declarations: used to process inference rules.
- Sub-typing. Used to define different “roles” for meta-data structures. For example you can say that an atomic value can be also used as an arithmetic expression.
- Rules.
- It is possible to embed types and methods from an external language.

- Premises containing bindings, clauses, function calls.
- Conclusion containing a function call.

Example:

```
y == a
bar k -> dataType c d
k := dataType c d
-------------------------
foo (dataType a b) y -> k
```

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

$$\text{R2: } \frac{\begin{array}{c} \forall c_i \in C \text{ , } \langle c_i \rangle \Rightarrow true \\ \forall f_j \in F \text{ , } \exists r_k \in R \mid \langle f_j^{r_k} \rangle \Rightarrow \{x_{r^k}\} \end{array}}{\langle f^{r} \rangle \Rightarrow \{x_r\}}$$

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

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

- We assume we have meta-data structures representing values in our language.
- We assume we already have defined expression evaluations for brevity.
- The memory is represented as a meta-data structures containing a map between Id's and values.

```
Data "$m" << ImmutableDictionary<Id, Value> >> :
    SymbolTable
```

- We represent local scopes through a list of symbol tables.

```
Data SymbolTable -> "::" -> TableList : TableList
```

Meta-data definition of If-Then-Else:

```
Data "then" : Then
Data "else" : Else
Data "if" -> Expr -> Then -> Stmt -> Else -> Stmt : Stmt
```

Meta-data definition of While-Do:

```
Data "do" : Do
Data "while" -> Expr -> Do -> Stmt : Stmt
```

Evaluation function:

```
Func "eval" -> TableList -> Stmt : EvaluationResult
```

Evaluation of If-Then-Else

```
evalExpr tables condition -> $b true
emptyDictionary -> table
eval (table :: tables) thenBlock -> table' :: tables''
-----------------------------------------------
eval tables (if condition then thenBlock else elseBlock) ->
    tables''
```

```
evalExpr tables condition -> $b false
emptyDictionary -> table
eval (table :: tables) elseBlock -> table' :: tables''
-----------------------------------------------
eval tables (if condition then thenBlock else elseBlock) ->
    tables''
```

- Pattern matching of the statement in the conclusion.
- Pattern matching of the first premise result.
- Create an empty symbol table for the if-then-else scope
- evaluate either the then or else.
- Return the state without the if-then-else scope.

Evaluation of While-Do

```
evalExpr tables condition -> $b false
-------------------------------------------
eval tables (while condition expr) -> tables
```

```
evalExpr tables condition -> $b true
emptyDictionary -> table
eval (table :: tables) block -> table' :: tables''
eval tables'' (while condition do block) -> res
--------------------------------------------------
eval tables (while condition do block) -> res
```

- Pattern matching of the statement in the conclusion.
- Pattern matching of the first premise result.
- If the condition returns true then create an empty symbol table. Otherwise skip the loop completely.
- Evaluate the body of the loop.
- Re-evaluate the whole loop (including the condition).
- Return the result of the previous step.

Advantages:

- Shorter code.
- Semantics almost identical to the formal formulation.

Disadvantages:

- Low performance due to the memory representation.
- Possible errors are reported at run-time.
- Languages implemented in Metacasanova exhibits dynamic behaviours.

- The state is represented through a meta-data structure in Metacasanova.
- Typing or executing the semantics requires to access a dictionary data structure at run-time.
- This is due to the fact that it is not possible to extend the **meta-type system** to embed the type system of the implemented language.

- We extend Metacasanova with functors (functions that process types instead of values) and Modules.
- Functors and Modules are processed at compile-time rather than run-time.
- They allow to embed the type system of the language that is being implemented in the meta-type system.
- We introduce the symbol => to denote something evaluated at compile-time, in contrast to ->, which evaluates something at run-time.

We now proceed to define an alternate memory model:

The meta-type of a record is defined through a module. This module contains a functor that returns the type of the record (we use $*$ for *kind*, which means any type).

```
Module "Record" : Record {
  Functor "RecordType" : *
}
```

A record can be implemented as a sequence of pairs containing the field name and its type

```
Functor "EmptyRecord" : Record
Functor "RecordField" => string => * => Record : Record
```

The empty record contains a constructor that returns unit.

```
--------------------
EmptyRecord => Record {

Func "cons" : unit

-------------------
RecordType => unit

-------------------
cons -> ()

}
```

A field contains a functor returning the type of the field and a constructor for the record that returns a tuple where the first element has the type of the current field and the second has the type of the rest of the record.

```
------------------
RecordField name type r = Record {
Func "cons" -> type -> r.RecordType : RecordType

---------------------------------------
RecordType => Tuple[type,r.RecordType]

-------------------
cons x xs -> (x,xs)}
```

This creates a record for a physical body with two fields:

```
Functor "PhysicalBodyType" : Record

EmptyRecord => empty
RecordField "Velocity" Vector2 empty => velocity
RecordField "Position" Vector2 velocity => body
--------------------------
PhysicalBodyType => body
```

The premises will generate three separate modules:

1. The empty record module seen above
2. A `Record` instantiation for the field velocity followed by the empty record containing:

```
Func "cons" -> Vector2 -> unit : Tuple[Vector2,unit]

------------------------
cons x xs -> (x,xs)
```

3. A `Record` instantiation for the field position followed by velocity containing:

```
Func "cons" -> Vector2 -> Tuple[Vector2,unit] :
  Tuple[Vector2,Tuple[Vector2,unit]]

--------------------
cons x xs -> (x,xs)
```

The physical body can then be constructed as

```
Func "PhysicalBody" : PhysicalBodyType.RecordType

------------------------
PhysicalBody ->
  PhysicalBodyType.cons (0,0)
    ((0,0),())
```

Getters and setters are also modules:

```
Module "Getter" => (name : string) => (r : Record) {
        Functor "GetType" : *
        Func "get" -> (r.RecordType) : GetType }
```

Case 1: the field is the current element of the tuple. get returns the first element of the tuple.

```
name = fieldName
thisRecord := RecordField name type r
-----------------
GetField fieldName (RecordField name type r) => Getter
    fieldName thisRecord {

        ----------------
        GetType => type

        ---------------
        get (x,xs) -> x}
```

Case 2: the field is not the current element of the tuple. get generates a getter module that will eventually fall in Case 1 (assuming that the field name is valid):

```
name <> fieldName
thisRecord := RecordField name type r
-------------------
GetField fieldName (RecordField name type r) => Getter
    fieldName thisRecord{
        Functor "GetAnotherField" : Getter

        ---------------
        GetAnotherField => GetField fieldName r

        GetAnotherField => g
        ---------------
        GetType => g.GetType

        GetAnotherField => getter
        getter.get xs -> v
        -------------------
        get (x,xs) -> v }
```

Table: Code length implementation

Statement	Metacasanova	C#
`if-then-else`	4	103
`while`	7	73
`For`	11	81

Casanova 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: Runtime performance

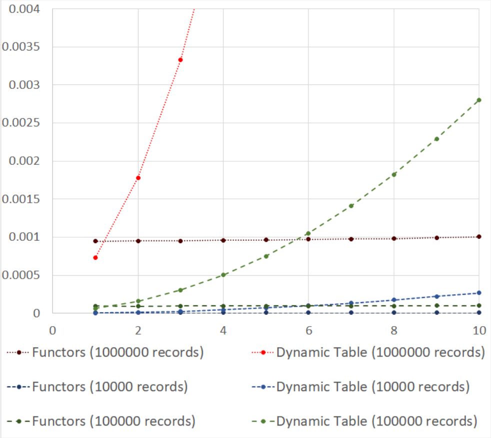
C--	Python
1.26ms	$2.36 \cdot 10^{-2}$ms

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: Running time with the functor optimization and the dynamic table with 1000000 records.

FIELDS	Functors (ms)	Dynamic Table (ms)	Gain
1	9.47E-04	7.29E-04	0.77
2	9.51E-04	1.78E-03	1.87
3	9.50E-04	3.33E-03	3.51
4	9.60E-04	5.43E-03	5.66
5	9.65E-04	8.03E-03	8.32
6	9.71E-04	1.11E-02	11.44
7	9.75E-04	1.47E-02	15.12
8	9.82E-04	1.89E-02	19.28
9	9.92E-04	2.37E-02	23.86
10	1.00E-03	2.87E-02	28.62
		Average gain	11.84

Figure: Performance chart



Benefits:

- Significant code reduction.
- Performance improvement and static typing with functors.
- Fast prototyping and implementation of new languages.

Problems:

- Programs in the implemented language still need to be expressed in the meta-language.
- Performance is worse than a hard-coded implementation of the compiler.

Future work:

- Use functors to extend Casanova, a DSL for game development, with networking primitives.
- Web-based meta-interpreter for a didactic platform to learn programming with interactive feedback.

Thank you!