Building game scripting DSL's with the Metacasanova metacompiler

Francesco Di Giacomo
Agostino Cortesi Mohamed Abbadi
Universita' Ca' Foscari, Venezia
francesco.digiacomo@unive.it
cortesi@unive.it
mohamed.abbadi@unive.it

Pieter Spronck
Tilburg University
p.spronck@uvt.nl

Giuseppe Maggiore
Hogeschool Rotterdam
maggg@hr.nl

Abstract

Many video games rely on a Domain Specific Language (DSL) to implement particular features such as artificial intelligence or time and synchronization primitives. Building a compiler for a DSL is a time-consuming task, and adding new features to a DSL is hard due to the low flexibility of the implementation choice. In this paper, we introduce an alternative to hand-made implementations of compilers for DSLs for game development: the Metacasanova metacompiler. We show the advantages of this metacomplier in terms of simplicity of designing and coding requirements, and in terms of performance of the resulting code, whose efficiency is comparable with hand-made implementations in commercial general purpose languages.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors—Translator writing systems and compiler generators

Keywords metacompiler, compiler generator, Domain specific language, game development

1. Introduction

In video games development it is often the case that *Domain Specific languages* (DSL) are used, as they provide ad-hoc features that simplify the coding process and yield to more concise and readable code when dealing with time management, synchronization, and AI thanks to their little CPU and memory overhead [1, 8, 21] . A typical synchronization problem arises when waiting for an event to happen, for instance when the short distance of a player from a door enables the player to open it. These scenarios usually happen in a heavily concurrent system, where possibly hundreds of entities perform such interactions within the same update of the game. In order to tackle these problems, as a valuable alternative to the use of Threads, Finite state machines, or Strategy patterns, developers make use of Domain specific languages, like JASS [3], Unreal Script [19], or NWScript [2].

A first approach is implementing a DSL by building an interpreter within the host language abstractions, such as monads in a functional programming language [10, 11, 18]. Unfortunately the performance of an interpreted DSL built with monads is not as high as that achieved by compiled code, as monads make a large use of anonymous functions (or lambda expressions) which are often implemented with virtual method calls. Moreover functional languages are rarely employed in game developments as games are highly stateful programs.

Another typical approach is to design a hard-coded compiler for the DSL. This is a hard and time-consuming task, since a compiler is made of several components which perform transformations from the source code into machine code. The steps performed in this transformations are often the same, regardless of the language for which the compiler is being implemented, and they are not part of the creative aspect of language design [4]. This is why metacompliers come into the scene, with the ability to treat programs as data [6].

In this paper we present a novel solution to ease the development of a compiler for a game DSL by developing a metacompiler producing code that is both clear and efficient, especially designed for games development.

The main differences between Metacasanova and traditional metacompilers are the following:

- Types for language structures defined in the metacompiler.
- Polymorphic language structures to specify multiple "roles" for them.
- Embed C# libraries and code into the language definition and use C# data types to define language structures.
- Rule branching, allowing the same expression to be evaluated in different ways returning all the possible evaluations.

In Section 2 we describe in detail the most common techniques to solve the concurrency problems in games. In Section 3 we propose a novel approach to defining DSL's for games by using a metacompiler called Metacasanova. We will explain informally the structure of a program in Metacasanova and then formalize the syntax and semantics. In Section 4 we will present as case study Casanova 2.5, a re-implementation of Casanova, a language oriented to video game development, in the matacompiler. We choose this particular language for its high-performance and usability in the scope of game development [1]. In Section 5 we compare the

25th International Conference on Compiler Construction (CC 2016), March 17–18, 2016, Barcelona, Spain.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

running time of the generated code of Casanova 2.5 for a sample with an analogous implementation in Python. Moreover we compare the code length of the implementation of Casanova 2.5 in Metacasanova with the implementation of the current Casanova hard-coded compiler. In Section 6 we will present further improvement that will be implemented in Metacasanova in the future.

1.1 Related work

In this section we present the existing work on metacompilers and Domain specific languages for games.

Metacompilers A first approach to metacompilers was first made with the Schorre-metalanguages family of metacompilers, among which we consider META-II [17]. A META-II program is a set of rules. Each rule can either succeed or fail. The body of the rule is made of *test*, *operators*, and *output code*. The rules allow to express the syntax of the language in a BNF-like syntax. Each rule is evaluated, i.e. the tests are performed, and if the tests pass then the output code is generated (in the specific case the output was machine code for the IBM 1401). An improvement to this metacompiler was TREE-META [16], which allows the definition of tree-building operators to generated an *Abstract Syntax Tree*. This feature allows the implementation of language optimizations on the AST. Finally, CWIC [4] was a further refinement, employing three sub-languages used to define the different stages of the transformation process: *Syntax* is used to defined the transformation between the grammar definition and the associated AST. *Generator* is made of a series of transforming rules to output IBM 360 binary code. *MOL-360* is an intermediate language which was introduced to write support libraries for CWIC and added later as an intermediate layer in the code generation phase. These compilers ease the process of defining and checking the grammar of the language, but they do not allow to define semantics rule (i.e. a type system). Indeed they simply parse the input language and then directly output the target code.

The type semantics and operational semantics are often expressed in the form of logic rules, which define transformations among AST's. For this reason a metacompiler can be seen as a set of logic rules defining these transformations. The RML metacompiler [5, 9, 15] follows this idea. A program in RML is made of a set of rules. Each rule is made of a set of *premises*, whose evaluation produces a result, and of a *conclusion* which produces the final result of the rule. Each of the premises is evaluated and can result in a fail or a success. Premises can be combined using logical operators, assuming that a success corresponds to *true* and a fail to *false*.

Domain specific languages for games A very common domain specific language is Unreal Script [19], used to develop a variety of commercial games such as Unreal, Unreal Tournament, and Deus Ex. It is an object-oriented language interpreted in the Unreal Engine. In this language it is possible to define particular functions called `states` to model the concurrent behaviours described above. These functions are executed only when the object is in that particular state.

Another widely used language for games is the script language created for the Neverwinter Nights game series, called NWScript. This language is an extension of the C programming language with extended native types to support in-game elements, such as graphics effects, characters, etc. This language does not provide primitives to explicitly interrupt the execution or synchronize the execution with other events in the games, rather it allows to run a specific script when a set of pre-defined events happen in the game.

The Game Maker Script language [14] is another widely used script language based on a syntax similar to C. It is used in the Game Maker game engine.

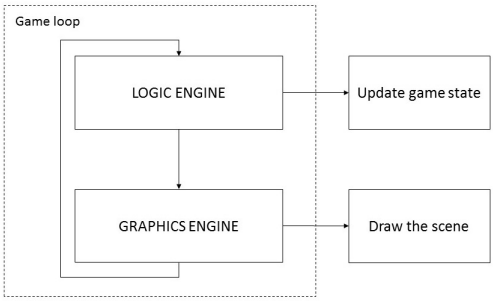

Figure 1. Game loop

A different approach in literature is the Game Description Language [20], a logic language that represents the game state as facts and the game mechanics as logical rules.

2. The challenges of building a game DSL

In this section we introduce the general architecture of a game. We then present an example of common timing and synchronization primitives used in DSL's for games and we show some techniques typically used to implement them. For each technique we list the main drawbacks. Finally we present our solution to the problem of developing a DSL for games.

2.1 Preliminaries

A game engine is usually made by several interoperating components. All the components use a shared data structure, called *game state*, for their execution. The two main components of a game are the *logic engine*, which defines how the game state evolves during the game execution, and the *graphics enginge*, which draws the scene by reading the updated game state. These two components are executed in lockstep within a function called *game loop*. The game loop is executed indefinitely, updating the game state by calling the logic engine, and drawing the scene by using the graphics engine. An iteration of the game loop is called *frame*. Usually a game should run between 30 to 60 frames per second. This requires both the graphics engine and the logic engine to be high-performance. In this paper we will only take into account the performance of the logic engine, as scripting drives the logic of the game loop. A schematic representation of this architecture can be seen in Figure 1.

2.2 A time and synchronization primitive

A common requirement in game DSL's is a statement which allows to pause the execution of a function for a specified amount of time or until a condition is met. We will refer to these statements as `wait` and `when`. Such a behaviour can be modelled using different techniques:

- *Threads* allow to solve such synchronization problems but they are unsuitable for video game development because of memory usage and CPU overhead due to their scheduling.
- *Finite State Machines* allow to represent such concurrent behaviours [1] by using a `switch` control structure to an opportune state. For instance in the case of timed wait, the states are *wating* and `clear` (when the timer has elapsed). This solution is high-performance but the logic of the behaviour is lost inside the `switch` structure.
- *Strategy pattern* allows to represent the instructions of the language has polymorphic data types. Each concurrent structure is

implemented by a class which defines the behaviour of the current structure, and the next structure to execute. This solution is not high-performance due to virtuality and the high number of object instantiations.

- *Monadic DSL* uses a variation of the state monad. It represents scripts with two states: *Done* and *Next*. The bind operator is used to simulate the code interruption. This approach is simple and elegant but it suffers of the same virtuality problems of the strategy pattern, this time because of the extensive use of lambda expressions.
- *Compiled DSL* is the most common solution and allows to represent the concurrency by using concurrent control structures defined in the language. Compiled DSL's grant high-performance and code readability, but they require to implement a compiler or an interpreter for it.

In what follows we show a possible implementation of these statements using the presented techniques. We deliberately ignore the thread-based implementation since, as explained above, threads are not suitable for game development.

Monadic DSL: A possible approach to building an interpreted DSL is using monads in a functional programming language, as done in [11]. A script is a function that takes no input parameters and returns the state of the script after the execution of the current statement. The state can be either *Done*, when the script terminates, or *Next* when the script is still running and we need to pass the rest of the script to be evaluated. We present a definition of the monad in the following code snippet in pseudo-ml:

```
type Script<'a> = Unit -> State<'a>
and State<'a> = Done of 'a | Next of Script<'a>
```

We can define the `Return` and `Bind` operators as follow:

```
let return (x : 'a) : Script<'a> =
  fun () -> Done x

let (p : Script<'a>) >>= (k : 'a -> Script<'b>) : Script<'b> =
  fun () ->
    match p with
    | Done x -> k x ()
    | Next p' -> Next(p' >>= k)
```

The `Return` operator simply returns a script that builds the result of the computation. The `Bind` operator checks the current state of the script: if the script has terminated then it simply builds the result by executing the last script statement, otherwise it returns the continuation containing the invocation of the `Bind` on the remaining part of the script. In this framework the `wait` statement can be implemented as follows (we assume that `do` is a shortcut for the bind on a function returning `Script<Unit>`):

```
let yield : Script<Unit> =
  fun () -> Next(fun () -> Done ())

let rec waitRecursion (interval : float32,
                       startingTime : float32) : Script<Unit> =
  let t = getTime()
  let dt = (t - t0)
  if dt < interval then
    do yield
    do waitRecursion(startingTime)
  else
    return ()

let wait (timer : float32) : Script<Unit> =
  let t0 = getTime()
  do waitRecursion(timer, t0)

let when (predicate : Unit -> bool) : Script<Unit> =
  if predicate () then return ()
  else
    do yield
    do when predicate
```

The `yield` function simply forces the script to stop for one frame. The `wait` function recursively checks if the timer has elapsed. If this is not the case it skips a frame and then re-evaluates otherwise it returns `Unit`. `wait` on a predicate simply skips a frame and keeps re-executing itself until the condition is met.

Strategy pattern Implementing `wait` and `when` with the strategy pattern requires to define an interface which contains the signature for a method to run the script. Usually scripts need to access the time elapsed between the current frame and the previous one and to a reference to the game state data structure for various reasons, which are passed as parameter for this method. The method returns the updated script after the current execution. We present the code for the interface in a pseudo-C# code:

```
public interface Script
  public Script Run(float dt, GameState state);
```

We will then model `wait` and `when` with two classes implementing such interface. Each of the script commands contains a reference to the next statement to execute.

```
public class Wait : Statement
  private Statement next;
  private float time;

  public Script Run(float dt, GameState state)
    if (time >= 0)
      time -= dt;
      return this;
    else
      return next;

public class When : Statement
  private Func<bool> predicate;
  private Script next;

  public Script Run(float dt, GameState state)
    if (predicate())
      return next;
    else
      return this;
```

Finite state machines: In this approach we model `wait` and `when` as finite state automata. Both statements require to store the state of the automata. `wait` requires to store a timer whose value is updated at each frame. `when` stores a predicate to execute and check at every game logic update. `wait` has three states: (*i*) timer initialization, (*ii*) check timer and update, and (*iii*) timer elapsed. In the first state we initialize the timer and then update it for the first time. The second state is used to check whether the timer has elapsed. The third state is used to continue with the execution after the time has elapsed.

```
//WAIT
public void Update(float dt, GameState gameState)
  switch(state)
    case -1:
      this.t = timer;
      state = 0;
      goto case 0;
    case 0:
      this.t = this.t - dt
      if (this.t <= 0)
        state = 1;
        goto case 1:
      else
        return;
    case 1:
    //run the code after wait
```

`when` has two states: (*i*) check predicate, (*ii*) predicate satisfied (go on with the execution). The first state checks if the predicate has been satisfied. If that is the case, we jump to the next state, otherwise we pause the execution and we remain in the same state.

```
//WHEN
public void Update(float dt, GameState gameState)
  switch(state)
    case 0:
      if (predicate())
        state = 1;
        goto case 1:
      else
        return;
    case 1:
    //run the code after when
```

Technique	Readability	Performance	Code length
Monadic DSL	✓	✗	✓
Strategy Pattern	✗	✗	✓
Finite state machines	✗	✓	✗
Hard-coded compiler	✓	✓	✗

Table 1. Pros and cons of script implementation techniques

Waiting with a hard-coded compiler: This approach requires to build a compiler by generating the syntax using a standard lexer/parser generator. After that usually it is required to define a set of type rules, and the operational semantics of the language constructs. The type rules and the operational semantics are then implemented in the type checker and the code generator of the compiler.

2.3 Discussion

In the previous paragraph we have seen different techniques employed by developers to implement timing and synchronization statements commonly used in video games. We now list the advantages and disadvantages of each solution:

- *Monadic DSL:* this solution is elegant but inefficient, because of the extensive use of lambda expressions in monads. Indeed lambdas are usually implemented with virtual method calls. The code to define a monadic DSL is compact.
- *Strategy Pattern:* this solution is simple but low-performance because of the virtual method calls the logic update must invoke to run the statements and the high number of object instantiation to generate the script statements (one per statement). Besides the readability of the program for long scripts is lost due to the long chain of object instantiations. A library supporting scripts implemented with the strategy pattern is compact.
- *Finite state machines:* this approach is high-performance due to the small overhead of the `switch` control structure. Unfortunately the logic of the program for very complex scripts with several nested synchronizations is lost inside huge `switch` structures and the complexity increases drastically with the number of interruptions and synchronizations in the script [13]. Moreover note that for each timer we have to maintain a separate timer, and for any of the two control structures we need to store the state of the automaton. Implementing complex synchronizations require long and complex state machines.
- *Hard-coded compiler:* this approach is high-performance, as we could generate the state machines presented above during the code generation step, but building, maintaining, and extending a hard-coded compiler is a hard and time-consuming task. The code length increases with the size of the compiler (in term of modules and code lines) and the language complexity.

This situation is summarized in Table 1.

In this work we propose another development approach in building a game DSL by using a metacompiler, a program which takes as input a language definition, a program written in that language, and generates executable code.
Given this considerations, we formulate the following problem statement.

PROBLEM STATEMENT: Given the formal definition of a game DSL our goal is to automate, by using a metacompiler, the process of building a compiler for that language in a (*i*) short (code lines), (*ii*) clear (code readability), and (*iii*) efficient (time execution) way, with respect to a hand-made implementation.

3. The Metacasanova metacompiler

In this section we show how `wait` and `when` can be expressed with type and semantics rules. We show how this rules are implemented in a hard-coded compiler. We then introduce the idea of the metacompiler, explaining the advantage over a hard-coded compiler. We give an informal overview of the Metacasanova metacompiler and we then proceed to formalize its syntax and semantics.

3.1 Type and semantics of wait and when

Usually the type and semantics rules of language elements are represented by rules that resemble those of logic models. Each rule is made of a set of *premises* and a *conclusion*.

$$\frac{\begin{array}{c} premise_1 \\ premise_2 \\ \dots \\ premise_n \end{array}}{conclusion}$$

The conclusion is true if all the premises are true. According to this model, the type rules for `wait` and `when` are the following ($E \vdash x \ : \ T$ means that x has type T in the environment E):

$$\frac{E \vdash t \ : \ \texttt{float}}{E \vdash \texttt{wait}\ t \ : \ \texttt{void}} \qquad \frac{E \vdash c \ : \ \texttt{bool}}{E \vdash \texttt{when}\ c : \ \texttt{void}}$$

while their operational semantics is (with $\langle expr \rangle$ we mean "evaluating exp", with ; a sequence of statements, and with dt the time difference between the current frame and the previous):

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

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

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

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

3.2 Implementation in a hard-coded compiler

The semantics rules of `wait` and `when` can be implemented into the type checker module of a compiler written in a general purpose language. We assume that these two statements are represented as a discriminated union in the language AST as shown by the following pseudo-ml code snippet:

```
type Statement =
| Wait of Expression
| When of Expression
...
```

The rules are evaluated by means of a recursive function. In the case of a `wait` statement, we first type check its argument. If the argument is a float than we return the node in the type-checked AST corresponding to the type-checked `wait`. If the argument has another type then we raise an exception since the argument has not a valid type. In the case of a `when` statement we do the same, but this time we check that the argument has boolean type.

```
let rec typecheckStatement ( stmt : Statement ) : TypedStatement =
match stmt with
  | Wait ( expr ) ->
    let exprType = typecheckExpr expr
    match exprType with
```

```
        | Float -> Wait ( FloatExpr )
        | _ -> failwith (" Expected Float but given " + exprType . ToString
              ())
  | When ( expr ) ->
        let exprType = typecheckExpr expr
        match exprType with
        | Boolean -> Wait ( BoolExpr )
        | _ -> failwith (" Expected Boolean but given " + exprType .
              ToString ())
...
```

The code generation part requires to output code according to the semantics rules defined above. In this step the compiler can, for example, generate state machines implementing the behaviour described in Section 2.2.

3.3 Motivation for Metacasanova

From the example above we can see that, regardless of the implemented language, the process of type checking and implementing the operational semantics in a hard-coded compiler, is repetitive. Indeed, building the type checker and the code generator of a hard-coded compiler is a single, fixed translation of these rules into the general purpose language that was chosen for the implementation. This process can be summarized by the following behaviour:

- Find a rule which conclusion matches the structure of the language we are analysing.
- Recursively evaluate all the premises in the same way.
- When we reach a rule with no premises (a base case), we generate a result (which might be the type of the structure we are evaluating or code that implements its operational semantics).

Our goal is to take this process and automate it, starting only from the specifications which the hard-coded compiler would implement.

In the following sections we define the requirements the *Metacasanova* metacompiler must satisfy. We present informally the structure of a program in Metacasanova. We then proceed in defining the formal syntax in BNF of Metacasanova grammar. Finally we define the semantics of Metacasanova.

3.4 Requirements of Metacasanova

In order to relieve programmers of manually defining the behaviour described above in the back-end of the compiler, we propose to use a metacompiler. This metacompiler must include the following features:

- It must be possible to define custom operators (or functions) and data containers. This is needed to define the syntactic structures of the language we are defining.
- It must be typed: each syntactic structure can be associated to a specific type in order to be able to detect meaningless terms (such as adding a string to an integer) and notify the error.
- It must be possible to have polymorphic syntactical structures. This is useful to define equivalent "roles" in the language for the same syntactical structure; for instance we can say that an integer literal is both a *Value* and an *Arithmetic expression*.
- It must natively support the evaluation of semantics rules, as those shown above. A *rule*,in Metacasanova, in the fashion of a logic rule, is made of a sequence of premises and a conclusion. The premises can be function calls or clauses. Clauses are boolean expressions that are checked in order to proceed with the rule evaluation. The function call will run in order all the rules that contain that function as conclusion. The return value of the first rule that succeeds is taken. A rule returns a value if all the clauses evaluate to `true` and all the function calls succeed.

From this specifications, we see that our goal is indeed a metacompiler, as it takes as input a language definition, a program for that language, and outputs runnable code that mimics the code that a hard-coded compiler would output.

3.5 General overview

A Metacasanova program is made of a set of `Data` and `Function` definitions, and a sequence of rules. A data definition specifies the constructor name of the data type (used to construct the data type), its field types, and the type name of the data. Optionally it is possible to specify a priority for the constructor of the data type. For instance this is the definition of the sum of two arithmetic expression

```
Data Expr -> "+" -> Expr : Expr  Priority 500
```

Note that Metacasanova allows you to specify any kind of notation for data types in the language syntax, depending on the order of definition of the argument types and the constructor name. In the previous example we used an infix notation. The equivalent prefix and postfix notations would be:

```
Data "+" -> Expr -> Expr : Expr
Data Expr -> Expr -> "+" : Expr
```

A function definition is similar to a data definition but it also has a return type. For instance the following is the evaluation function definition for the arithmetic expression above:

```
Func "eval" -> Expr : Evaluator => Value
```

In Metacasanova it is also possible to define polymorphic data in the following way:

```
Value is Expr
```

In this way we are saying that an atomic value is also an expression and we can pass both a composite expression and an atomic value to the evaluation function defined above.

Metacasanova also allows to embed C# code into the language by using double angular brackets. This code can be used to embed .NET types when defining data or functions, or to run C# code in the rules. For example in the following snippets we define a floating point data which encapsulates a floating point number of .NET to be used for arithmetic computations:

```
Data "$f" -> <<float>> : Value
```

A rule in Metacasanova, as explained above, may contain a sequence of function calls and clauses. In the following snippet we have the rule to evaluate the sum of two floating point numbers:

```
eval a => $f c
eval b => $f d
<<c + d>> => res
-------------------------
eval (a + b) => $f res
```

Note that if one of the two expressions does not return a floating point value, then the entire rule evaluation fails. Also note that we can embed C# code to perform the actual arithmetic operation. Metacasanova selects a rule by means of pattern matching in order of declaration on the function arguments. This means that both of the following rules will be valid candidates to evaluate the sum of two expressions:

```
...
---------------
eval expr => res

...
-----------------
eval (a + b) => res
```

In Metacasanova it is also possible to branch rule evaluation if more than one rule is suitable to evaluate the function call. In this case we will generate a list of results based on all the possible

execution branches. In order to achieve this we use the operator **==>** instead of **=>**.

Finally the language supports expression bindings with the following syntax:

```
x := $f 5
```

3.6 Syntax in BNF

The following is the syntax of Metacasanova in Backus-Naur form. Note that, for brevity, we omit the definitions of typical syntactical elements of programming languages, such as literals or identifiers:

```
<program> ::=
  {<include>} {<import>} {<data>} <function> {<function>} {<alias>} <rule>
        {<rule>}
<premise> ::=
  <clause> | <functionCall> | <binding>
<binding> ::=
  id ":=" <constructor>
<rule> ::=
  {premise} "-" {"-"} <functionCall>
<clause> ::= //typical boolean expression
<functionCall> ::=
  <id> {<argument>} <arrow> <argument> |
  {<argument>} <id> {<argument>} <arrow> <argument> |
  <id> {<argument>} <arrow> <argument>
<arrow> ::= "=>" | "==>"
<constructor> ::=
  <id> {<argument>} |
  {<argument>} <id> {<argument>} |
  {<argument>} <id>
<external> ::= "<<" <csharpexpr> ">>"
<csharpexpr> ::= //all available C# expressions
<argument> ::=
  ["("]
    (<id> |
    <external> |
    <literal> |
    <constructor>)
  [")"]
<literal> ::= //typical literals such as integer, float, string, ...
<import> ::= import id {"." id}
<include> ::= include id {.id}
<alias> ::= <typeDef> is <typeDef>
<typeDef> ::= id | "<<" id ">>"
<typeArguments> :: =
  '"' <id> '"' {"->" <typeDef>} ":" <typeDef> |
  <typeDef> {"->" <typeDef>} "->" '"' <id> '"' {"->" <typeDef>} ":" <
        typeDef> |
  <typeDef> {"->" typeDef} "->" '"' <id> '"' ":" <typeDef>
<function> ::= Func <typeArguments> "=>" <typeDef> [Priority <literal>]
<data> ::= Data <typeArguments> [Priority <literal>]
```

3.7 Semi-formal Semantics

In what follows we assume that the pattern matching of the function arguments in a rule succeeds, otherwise a rule will fail to return a result. The informal semantics of the rule evaluation in Metacasanova is the following:

R1 A rule with no clauses or function calls always returns a result.

R2 A rule returns a result if all the clauses evaluate to `true` and all the function calls in the premise return a result.

R3 A rule fails if at least one clause evaluates to `false` or one of the function calls fails (returning no results).

We will express the semantics, as usual, in the form of logical rules, where the conclusion is obtained when all the premises are true. In what follows we consider a set of rules defined in the Metacasanova language R. Each rule has a set of function calls F and a set of clauses (boolean expressions) C. We use the notation f^r to express the application of the function f through the rule r. We will define the semantics by using the notation $\langle expr \rangle$ to mark the evaluation of an expression, for example $\langle f^r \rangle$ means evaluating the application of f through r. Note that the result of evaluating f through r produces generally a set of results because of the branching operator described above. In the base case R1 we return a single result because a rule without premises cannot branch. The following is the formal semantics of the rule evaluation in Metacasanova, based on the informal rules defined above:

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

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

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

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

R1 says that, when both C and F are empty (we do not have any clauses or function calls), the rule in Metacasanova returns a result. R2 says that, if all the clauses in C evaluates to true and, for all the function calls in F we can find a rule that returns a result (all the function applications return a result for at least one rule of the program), then the current rules return a result. R3(a) and R3(b) specify when a rule fails to return a result: this happens when at least one of the clauses in C evaluates to false, or when one of the function applications does not return a result for any of the rules defined in the program.

4. Case study: Casanova 2.5, a language for game development

In this section we will briefly introduce the Casanova language, a domain specific language for games. We than show a re-implementation, which we call Casanova 2.5, of the Casanova 2 language hard-coded compiler as an example of use of Metacasanova.

4.1 The Casanova language

Casanova 2.5 is a language oriented to video game development which is based on Casanova 2 [1]. A program in Casanova is a tree of *entities*, where the root is marked in a special way and called *world*. Each entity is similar to a *class* in an object-oriented programming language: it has a constructor and some fields. For instance the following code snippet shows an entity depicting a movable character in a 2D strategy game:

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

  ...

  Create(p : Vector2) = {
    Position = new Vector2(3.0f, 5.0f)
    Velocity = Vector2.zero
  }
}
```

The fields do not have access modifiers because they are not directly modifiable from the code except with a specific statement. Each entity also contains a list of *rules*, that are methods that are ticked in order with a specific refresh rate called `dt`. Each rule takes as input four elements: `dt`, `this`, which is a reference to the current entity, `world` that is a reference to the world entity, and a subset of entity fields called *domain*. A rule can only modify the fields contained in the domain. The rules can be paused for a certain amount of seconds or until a condition is met by using the `wait` statement. It is possible to modify the values of the fields in the domain by using the `yield` statement which takes as input a tuple of values to assign to the fields. When the `yield` statement is executed the rule is paused until the next frame. Also the body of control structures (`if-then-else`, `while`, `for`) is interruptible.

For example the following rules are used to move the `Character` when pressing the appropriate key

```
entity Character = {
  ... // fields
  //move up
  rule Position =
    wait Input.GetKey(W)
    yield Position + (new Vector2(0.0f, -3.0f)) * dt
  //move left
  rule Position =
    wait Input.GetKey(A)
    yield Position + (new Vector2(-3,0f, 0.0f)) * dt
}
```

The rules update the `Position` using the Euler approximation of the differential equation ($p(t)$ and $v(t)$ are respectively the position and velocity at time t)

$$v(t) = \frac{dp(t)}{dt}$$

where we take the time elapsed between the current frame and the previous one as integration step. Each rule executes the position update only if the appropriate key is pressed.

In the following section we show the implementation of Casanova 2.5 in Metacasanova.

4.2 Casanova 2.5

The memory in Casanova 2.5 is represented using three maps, where the key is the variable/field name, and the value is the value stored in the variable/field. The first dictionary represents the global memory (the fields of the `world` entity or *Game State*), the second dictionary represents the current entity fields, and the third the variable bindings local to each rule.

The core of the entity update is the `tick` function. This function evaluates in order each rule in the entity by calling the `evalRule` function. This function executes the body of the rule and returns a result depending on the set of statements that has been evaluated. This result is used by `tick` to update the memory and rebuild the rule body to be evaluated at the next frame. The result of `tick` is a `State` containing the rules updated so far, and the updated entity and global fields. Since a rule must be restarted after the whole body has been evaluated, we need to store a list containing the original rules, which will be restored when evaluation returns `Done` (see below). At each step the function recursively calls itself by passing the remaining part of original rules (the rules which body was not altered by the evaluation of the statements) and modified rules (which body has been altered by the evaluation of the statements) to be evaluated. The function stops when all the rules have been evaluated, and this happens when both the original and the modified rule lists are empty.

Interruption is achieved by using *Continuation passing stlye*: the execution of a sequence of statements is seen as a sequence of steps that returns the result of the execution and the remaining code to be executed. Every time a statement is executed we rebuild a new rule whose body contains the continuation which will be evaluated next. For example, consider the following rule:

```
rule X,Y =
  while X > 0 do
    wait 1.0f
    yield X - 1,Y + 1
```

The code is executed atomically until the `wait` statement (assuming that the `while` condition is true). At that point we rebuild a new rule containing the code to execute at the next iteration:

```
rule X,Y =
  wait (1.0f - dt)
  yield X - 1, Y + 1
  while X > 0 do
    wait 1.0f
    yield X - 1,Y + 1
```

Note that the `while` is placed at the end of the continuation because it must be re-evaluated after the first iteration is complete, and

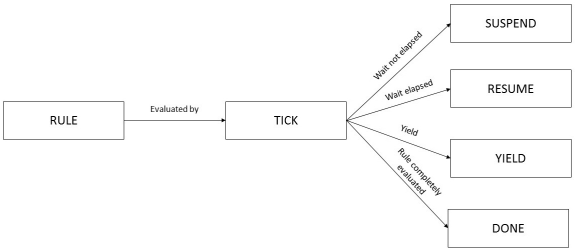

Figure 2. Casanova 2.5 rule evaluation

that we have decreased the waiting time by `dt` (the time elapsed between one frame and the previous one).

The possible results returned by the `tick` function are the following:

- `Suspend` contains a `wait` statement with the updated timer, the continuation, and a data structure called `Context` which contains the updated local variables, the entity fields, and the global fields. The function rebuilds a rule which body is the sequence of statements contained by the `Suspend` data structure.

```
evalRule (rule dom body k locals delta) fields globals =>
  Suspend (s;cont) (Context newLocals newFields newGlobals)
r := rule dom s cont newLocals dt
tick originals rs newFields newGlobals dt =>
  State updatedRules updatedFields updatedGlobals
st := State (r::updatedRules) updatedFields updatedGlobals
-------------------------------------------------------
tick (original::originals) ((rule dom body k locals delta)::rs) fields
      globals dt => st
```

- `Resume` is returned when the timer must resume after the last waited frame. In order not to skip a frame we must still re-evaluate the rule at the next frame and not immediately. In this case the argument of `Resume` is only the remaining statements to be executed.

```
evalRule (rule dom body k locals delta) fields globals =>
  Resume cont (Context newLocals newFields newGlobals)
r := rule dom cont nop newLocals dt
tick originals rs newFields newGlobals dt =>
  State updatedRules updatedFields updatedGlobals
st := State (r::updatedRules) updatedFields updatedGlobals
-------------------------------------------------------
tick (original::originals) ((rule dom body k locals delta)::rs) fields
      globals dt => st
```

- `Yield` stops evaluation for one frame. We use the continuation to rebuild the rule body. Memory is updated by `evalRule`.

```
evalRule (rule dom body k locals delta) fields globals =>
  Yield cont values (Context newLocals newFields newGlobals)
r := rule dom cont nop newLocals dt
tick originals rs newFields newGlobals dt =>
  (State updatedRules updatedFields updatedGlobals)
st := State (r::updatedRules) updatedFields updatedGlobals
-------------------------------------------------------
tick (original::originals) ((rule dom body k locals delta)::rs) fields
      globals dt => st
```

- `Done` stops the evaluation for one frame and rebuilds the original rule body by taking it from the original rules list.

```
evalRule r fields globals =>
  Done (Context newLocals newFields newGlobals)
tick originals rs newFields newGlobals dt =>
  State updatedRules updatedFields updatedGlobals
st := State (original::updatedRules) updatedFields updatedGlobals
-------------------------------------------------------------------
tick (original::originals) (r::rs) fields globals dt => st
```

You can see a schematic representation of the tick function in Figure 2 and Figure 3.

The function `evalRule` calls `evalStatement` to evaluate the first statement in the body of the rule passed as argument. The result of the evaluation of the statement is processed in the following way:

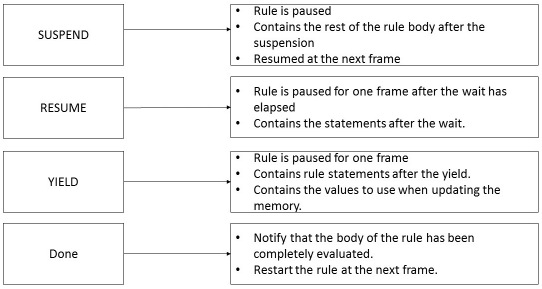

Figure 3. Results of rule evaluation

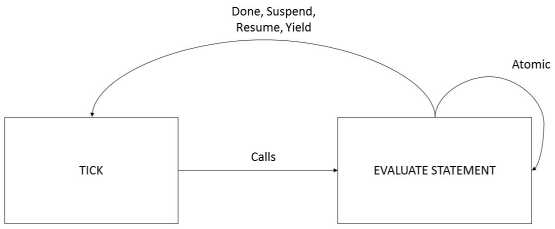

Figure 4. Statement evaluation

- If the result is `Done`, `Suspend` or `Resume` then it is just returned to the caller function.

```
evalStatement b k (Context locals fields globals) dt =>
  Done context
-----------------------------------------------
evalRule (rule dom b k locals dt) fields globals => Done context

evalStatement b k (Context locals fields globals) dt =>
  Suspend ks context
-------------------------------------------------------------------
evalRule (rule dom b k locals dt) fields globals  => Suspend ks context

evalStatement b k (Context locals fields globals) dt =>
  Resume ks context
-------------------------------------------------------------------
evalRule (rule dom b k locals dt) fields globals  => Resume ks context
```

- If the result is `Atomic` it means that the evaluated statement was uninterruptible and the remaining statements of the rule must be re-evaluated immediately.

```
evalStatement b k ctxt dt => Atomic z c
evalRule (rule dom z nop c dt) => res
--------------------------------
evalRule (rule dom b k ctxt dt)  => res
```

- If the result is `Yield` then the the fields in the domain are updated recursively in order and then the updated memory is encapsulated in the `Yield` data structure and passed to the caller function.

```
evalStatement b k (Context locals fields globals) dt =>
  Yield ks values context
updateFields dom values context  => updatedContext
-----------------------------------------------------------
evalRule (rule dom b k locals dt) fields globals => Yield ks values
      updatedContex
```

Note that, in case of a rule containing only atomic statements, we will eventually return `Done` after having recursively called `evalStatement` for all the statements, and the rule will be paused for one frame. A schematic representation of the function behaviour is depicted in Figure 4.

The `evalStatement` function is used both to evaluate a single statement and a sequence of statements. When evaluating a sequence of statements, the first one is extracted. A continuation is built with the following statement and passed to a recursive call to `evalStatement` which evaluates the extracted statement. If the existing continuation is non-empty, then it is added before the current continuation. If both the continuation and the body are empty (situation represented by the `nop` operator) then it means the rule evaluation has been completed and we return `Done`.

```
a != nop
----------------------
addStmt a b => a;b

--------------------
addStmt nop nop => nop

addStmt b k => cont
evalStatement a cont ctxt dt => res
--------------------------------
evalStatement (a;b) k ctxt dt => res

--------------------------------
evalStatement nop nop ctxt dt => Done ctxt
```

We will now present, for brevity, only the evaluation of the `wait` and `yield` statements. Both the evaluation of the control structures and the variable bindings always return `Atomic` because they do not, by definition, pause the execution of the rule.

The `wait` statement has two different evaluations, based on the rules defined in Section 2:

- The timer has elapsed: in this case we return `Resume` which contains the code to execute after the `wait` statement.
- The timer has not elapsed: in this case we return `Suspend` which contains the `wait` statement with the updated timer followed by the continuation.

```
<<t <= dt>> == false
-----------------------------------
evalStatement (wait t) k ctxt dt => Suspend wait <<t - dt>>;k ctxt

<<t <= dt>> == true
-----------------------------------
evalStatement (wait t) k ctxt dt => Resume k ctxt
```

The `yield` statement takes as argument a list of expressions whose values are used to update the corresponding fields in the rule domain. The evaluation rule recursively evaluates the expressions and stores them into a list passed as argument of the `Yield` result. Those arguments are used later by `evalRule` to update the corresponding fields.

```
-------------------------
evalYield nil ctxt => nil

eval expr ctxt => v
evalYield exprs ctxt => vs
-----------------------------------------
evalYield (expr :: exprs) ctxt => v :: vs
```

5. Evaluation

In this section we provide an implementation of a patrol script for an entity in a game. The sample is made up of an entity, representing a guard, and a couple of checkpoints. The guard continuously moves between the two checkpoints. We choose this sample because this is a typical behaviour implemented in several games, where the user is able to set up a patrol route for a unit. Below we provide the code for the patrol logic in a "prettified" version of Casanova 2.5, that is, using a syntax to get rid of all the additional structures used by the metacompiler for the rule evaluation. This syntax can be made available by using a parser which then

produces either the corresponding syntax in the metacompiler or directly the metacompiler AST. We choose to ignore this step because its solution is known in literature and trivial.

```
rule Position = Position + Velocity * dt

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

In the following section we show the comparison between the sample implemented in Casanova 2.5 and an equivalent implementation in Python with respect to the running time. We then show a comparison between the hard-coded compiler of Casanova 2.0 and the implementation of Casanova 2.5 in Metacasanova with respect to the code length. We want to underline that the main goal of this work is **to ease the process of building a compiler for a DSL for games, thus our main objective is decreasing the code length and complexity necessary to implement a hard-coded compiler for the language**. At the same time we show that the compiled program in Casanova 2.5 **has performance similar to that of a language used in game development, and thus Casanova 2.5 is usable in a real scenario**.

5.1 Chosen languages

We compared the running time of the sample in metacompiled Casanova with an equivalent implementation in Python. This language was chosen based on its use in game development: Python has been used extensively in several games such as Civlization IV [7] or World in Conflict [12] because of the native support for coroutines. Furthermore we decided on purpose not to compare the performance of the same game in C++ and C#, although they are widely used in the industry, because we knew in advance [1] that the current version of the code generated by the meta-compiler would not match the high performance of these languages. The sample has been run with a number of entities ranging from 100 to 1000 for 100 game state updates. An average update time was determined afterwards based on the total running time of the game logic update.

5.2 Performance

The performance results are shown in Table 2 and Figure 5. We see that the generated code has performance on the same order as Python. This is mainly due to the fact that the memory, in the metacompiled implementation of Casanova, is managed through a map, and because of the virtuality of the implemented operators. Each time Casanova accesses a field in an entity this must be looked up into the map. To this we add the complexity of dynamic lookups when we must deal with polymorphic results into the rules.

From Table 3 we see that the implementation of Casanova 2.0 language in Metacasanova is almost 5 times shorter in terms of lines of code than the previous Casanova implementation in F#. We believe it is worthy of notice that structures with complex behaviours, such as *wait* or *when*, require hundreds of lines of codes with a standard approach (the code lines to define the behaviour of the structure plus the support code to correctly generate the state machine), while in the meta-compiler we just need tens of lines of codes to implement the same behaviour. Moreover we want to point out that the previous Casanova compiler was written in a functional programming language: this languages tend to be more synthetic than imperative languages, so the difference with the same compiler implemented in languages such as C/C++ might be even greater.

The readability with respect to the hard-coded compiler code is also improved: we managed to implement the behaviour of synchronization and timing primitives almost imitating one to one the formal semantics of the language definition. In the hard-coded compiler implementation for Casanova 2.0 the semantics is lost in the code for generating finite state machines.

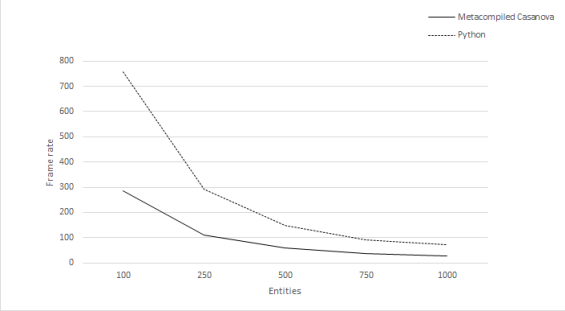

Figure 5. Performance comparison

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 2. Patrol sample evaluation

6. Conclusion and future work

In this work we proposed an alternative technique to implement a DSL for games by using a metacompiler called Metacasanova. As a case study we re-implemented the Casanova language, a DSL for game development, in Metacasanova. Our results show that the code required to re-implement Casanova in Metacasanova is (*i*) shorter, and (*ii*) more readable with respect to the existing hard-coded compiler for the same language. Moreover we showed that the language behaviour can be expressed in a way that directly mimics the formal semantics definition of the language. Adding the layer of the meta-compiler to the language affects the performance of the generated code so that we cannot achieve the same performance as with the manual implementation. Despite this, we managed to achieve performance similar to Python, a language typically used as a scripting language to define the game logic in several commercial games. We believe that this is a promising result for a meta-compiler which is at its first iteration.

Currently Metacasanova meta-compiler misses some features, such as a much stronger type system of modules, functions, higher kinded modules. This will allow to get rid of the memory map in Casanova 2.5 overhead by inlining fields directly into the rule updates. Furthermore we want to integrate networking primitives in Casanova 2.5 to ease the development of multiplayer games.

Despite this, adding additional features to Casanova in the implementation in Metacasanova requires little or no effort compared to what should be implemented in the hard-coded compiler that was developed for Casanova 2, at the same time generating code with performance good enough that could be used in a real scenario.

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 3. meta-compiler vs standard compiler

References

[1] M. Abbadi, F. Di Giacomo, A. Cortesi, P. Spronck, G. Costantini, and G. Maggiore. Casanova: A simple, high-performance language for game development. In S. Gbel, M. Ma, J. Baalsrud Hauge, M. F. Oliveira, J. Wiemeyer, and V. Wendel, editors, *Serious Games*, volume 9090 of *Lecture Notes in Computer Science*, pages 123–134. Springer International Publishing, 2015. ISBN 978-3-319-19125-6. . URL `http://dx.doi.org/10.1007/978-3-319-19126-3_11`.

[2] Bioware. Nwscript api reference. `http://www.nwnlexicon.com/`, 2002.

[3] Blizzard Entertainment. Jass api reference. `http://jass.sourceforge.net/doc/`, 1999.

[4] E. Book, D. V. Shorre, and S. J. Sherman. The cwic/36o system, a compiler for writing and implementing compilers. *SIGPLAN Not.*, 5 (6):11–29, June 1970. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/954344.954345`.

[5] D. Broman, P. Fritzson, G. Hedin, and J. Akesson. A comparison of two metacompilation approaches to implementing a complex domain-specific language. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1919–1921, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. . URL `http://doi.acm.org/10.1145/2245276.2232092`.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-30977-7.

[7] Firaxis Games. Civlization iv scripting api reference. `http://wiki.massgate.net/Our_Python_files_and_Event_Structure`, October 2008.

[8] J. P. Kelly, A. Botea, and S. Koenig. Offline planning with hierarchical task networks in video games. In *AIIDE*, 2008.

[9] D. Kgedal and P. Fritzson. Generating a modelica compiler from natural semantics specifications. In *In Proceedings of the Summer Computer Simulation Conference*, 1998.

[10] G. Maggiore, M. Bugliesi, and R. Orsini. Monadic scripting in f# for computer games. In *TTSS115th International Workshop on Harnessing Theories for Tool Support in Software*, page 35, 2011.

[11] G. Maggiore, A. Spanò, R. Orsini, M. Bugliesi, M. Abbadi, and E. Steffinlongo. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 287–292, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1168-7. . URL `http://doi.acm.org/10.1145/2305484.2305533`.

[12] Massive Entertainment. World in conflict script reference. `http://civ4bug.sourceforge.net/PythonAPI/`, September 2007.

[13] I. Millington and J. Funge. *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009. ISBN 0123747317, 9780123747310.

[14] M. Overmars. Game maker scripting reference. `http://docs.yoyogames.com/source/dadiospice/002_reference/index.html`, 1999.

[15] A. Pop and P. Fritzson. Debugging natural semantics specifications. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 77–82, New York, NY, USA, 2005. ACM. ISBN 1-59593-050-7. . URL `http://doi.acm.org/10.1145/1085130.1085140`.

[16] F. W. Schneider and G. D. Johnson. Meta-3 syntax-directed compiler writing compiler to generate efficient code. In *Proceedings of the 1964 19th ACM National Conference*, ACM '64, pages 41.501–41.508, New York, NY, USA, 1964. ACM. . URL `http://doi.acm.org/10.1145/800257.808898`.

[17] D. V. Schorre. Meta ii a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM National Conference*, ACM '64, pages 41.301–41.3011, New York, NY, USA, 1964. ACM. . URL `http://doi.acm.org/10.1145/800257.808896`.

[18] T. Sheard, Z. el-abidine Benaissa, and E. Pasalic. Dsl implementation using staging and monads. In *In Second Conference on Domain-Specific Languages (DSL'99*, pages 81–94. ACM, 1999.

[19] T. Sweeney and M. Hendriks. Unrealscript language reference. *Epic MegaGames, Inc*, 1998.

[20] M. Thielscher. A general game description language for incomplete information games, 2010.

[21] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/352029.352035`.