# Building game scripting DSL's with the Metacasanova metacompiler

No Author Given

No Institute Given

**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.

## 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 [14, 8, 1] . A typical synchronization problem arises when 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 [13], 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 [12, 10, 9]. 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 [5].

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

In this work we briefly describe the most common techniques used to build DSLs for game and their drawbacks (Section 2). We then propose a novel approach by introducing Metacasanova as a tool to develop a DSL (Section 3) and by re-emplementing Casanova DSL (Section 4). We then evaluate the result in terms of time performance and code length (Section 5) and draw the conclusion.

## 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.

### 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: (*i*) *Threads* can model this behaviour but are not suitable for game development due to their CPU and memory overhead, (*ii*) *Finite State Machines* are high performance but the code logic is lost inside a `switch` structure, (*iii*) *Strategy pattern* uses polymorphism to represent the language constructs but it is inefficient due to the extensive use of virtuality, (*iv*)

Monadic DSLs use monads to model the waiting or synchronization behaviour but extensively use virtuality as well due to lambda expressions, (*v*) *Compiled DSLs* are the most common solution, are high performance, but they require to implement a compiler or an interpreter.

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

**Table 1.** Pros and cons of script implementation techniques

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 then give an overview of how a program in Metacasanova is written.

### 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*. 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):

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

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

### 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. 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 Abstract Syntax Tree (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. 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 described in Section 2.2.

### 3.3 Motivation for Metacasanova

From the discussion above we observe 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: (*i*) find a rule which conclusion matches the structure of the language we are analysing, (*ii*) recursively evaluate all the premises in the same way, (*iii*) 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 order to achieve this we propose to use Metacasanova metacompiler. In what follows we show how a Metacasanova program is defined.

### 3.4 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
```

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.

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 (`$ f` is `Data` type for floating point values):

```
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. The code between angular brackets specifies C# code that can be embedded in Metacasanova, allowing to perform the arithmetic operations with .NET operators. 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
```

Finally the language supports expression bindings with the following syntax:

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

## 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. 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. 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.

The possible results returned by the `tick` function are the following: (*ii*) `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. (*ii*)`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. (*iii*) `Yield` stops evaluation for one frame. We use the continuation to rebuild the rule body. Memory is updated by `evalRule`. (*iv*)`Done` stops the evaluation for one frame and rebuilds the original rule body by taking it from the original rules list.

For brevity we write only the code for `Suspend`. A full implementation can be found at [7]. You can see a schematic representation of the tick function in Figure 1.

```
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
```

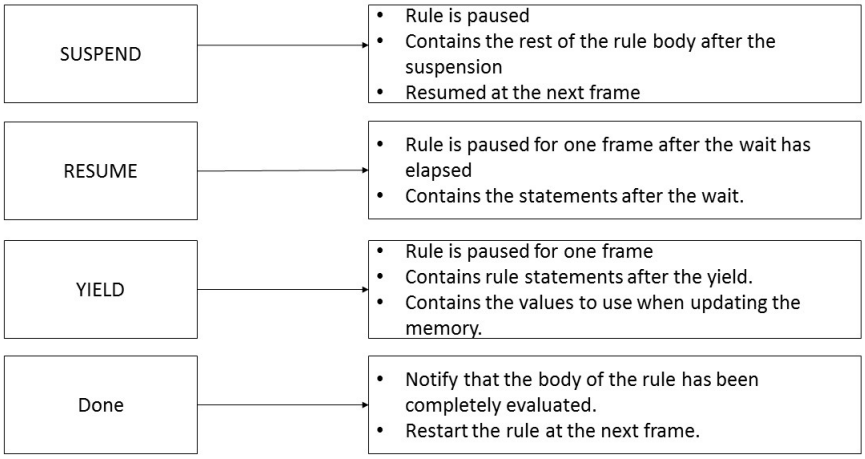


**Fig. 1.** Casanova 2.5 rule evaluation

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: (*i*) if the result is `Done`, `Suspend` or `Resume` then it is just returned to the caller function. We omit the code for this case, since it is trivial; (*ii*) 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; (*iii*) 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 ctxt dt => Atomic z c
evalRule (rule dom z nop c dt) => res
-------------------------------
evalRule (rule dom b k ctxt dt)  => res
```

```
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.
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: (*i*) the timer has elapsed: in this case we return `Resume` which contains the code to execute after the `wait` statement, or (*ii*) 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.

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

## 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. 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.

### 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 [6] or World in Conflict [11] because of the native support for coroutines. We deliberately ignore C++ and C# implementations, 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 main goal of this work is to reduce the effort of writing a compiler for a DSL for games while having acceptable performance.

### 5.2 Performance

The performance results are shown in Table 2. 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.

| **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

| **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

## 6 Conclusion

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.

## References


1. Mohamed Abbadi, Francesco Di Giacomo, Agostino Cortesi, Pieter Spronck, Giulia Costantini, and Giuseppe Maggiore. Casanova: A simple, high-performance language for game development. In Stefan Gbel, Minhua Ma, Jannicke Baalsrud Hauge, Manuel Fradinho Oliveira, Josef Wiemeyer, and Viktor Wendel, editors, *Serious Games*, volume 9090 of *Lecture Notes in Computer Science*, pages 123–134. Springer International Publishing, 2015.
2. Bioware. Nwscript api reference. `http://www.nwnlexicon.com/`, 2002.
3. Blizzard Entertainment. Jass api reference. `http://jass.sourceforge.net/doc/`, 1999.
4. Erwin Book, Dewey Val Shorre, and Steven J. Sherman. The cwic/36o system, a compiler for writing and implementing compilers. *SIGPLAN Not.*, 5(6):11–29, June 1970.
5. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
6. Firaxis Games. Civlization iv scripting api reference. `http://wiki.massgate.net/Our_Python_files_and_Event_Structure`, October 2008.
7. Francesco Di Giacomo. Casanova 2.5 source code. `https://github.com/vs-team/metacompiler/tree/master/Sources/Content/Content/CNV3`, 2016.
8. John Paul Kelly, Adi Botea, and Sven Koenig. Offline planning with hierarchical task networks in video games. In *AIIDE*, 2008.
9. 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.
10. Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico 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.
11. Massive Entertainment. World in conflict script reference. `http://civ4bug.sourceforge.net/PythonAPI/`, September 2007.
12. Tim Sheard, Zine el-abidine Benaissa, and Emir Pasalic. Dsl implementation using staging and monads. In *In Second Conference on Domain-Specific Languages (DSL'99*, pages 81–94. ACM, 1999.
13. Tim Sweeney and Michiel Hendriks. Unrealscript language reference. *Epic MegaGames, Inc*, 1998.
14. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.