Metacasnova: an optimized meta-compiler for Domain-Specific Languages

Francesco Di Giacomo, Agostino Cortesi
Università Ca' Foscari
Email: francesco.digiacomo@unive.it,cortesi@unive.it

Pieter Spronck, Mohamed Abbadi,
Giuseppe Maggiore
Tilburg University, Hogeschool Rotterdam
Email: p.spronck@tilburguniversity.edu,
abbam@hr.nl,
giuseppemag@gmail.com

***Abstract*—Implementing Domain-Specific Languages (DSL's) is desirable in several situations, as they offer language-level abstractions, which General-purpose languages do not offer, that speed up the implementation of the solution of problems limited to a specific domain. Developers are left with the choice of developing a DSL by building an interpreter/compiler for it, which is a hard an time-consuming task, or embedding it in a host language, thus speeding up the development process but losing several advantages that having a compiler might bring. In this work we present a meta-compiler called *Metacasanova*, which meta-language is based on operational semantics, with a further optimization based on functors. We show how this optimization leads to better performance by re-implementing Casanova, a DSL for game development, and by comparing it with its old, unoptimized implementation.**

I. Introduction

Domain-Specific Languages (DSL's) are becoming more and more relevant in software engineering thanks to their ability to provide abstractions at language level to target specific problem domains [12], [13]. Notable examples of the use of DSL's are (*i*) game development (UnrealScript, JASS, Status-quo, NWNScript), (*ii*) Database programming and design (SQL, LINQ), and (*iii*) numerical analysis and engineering (MATLAB, Octave). Two main alternatives have been proposed for the development of DSL's: the (*i*) *Embedding* technique, and (*ii*) the *Interpretation/Compilation* technique [10].

The former approach extends an existing programming language with the additional abstractions of the DSL. This is the case, for instance, of NWNScript, a scripting language for the Neverwinter Nights game extending the C language, and LINQ, which offers SQL-like abstractions extending C#. This technique has the advantage that the infrastructure of the host language can be widely re-used, thus reducing the development effort. Moreover, people expert on the host language can become proficient with the DSL extension in a short time. The disadvantages are that the syntax is likely to be far from that of the DSL formal syntax, since General-purpose languages generally do not offer syntax extensions, and Domain-specific optimization are difficult to achieve [8], [11].

The latter approach requires to develop an interpreter or compiler for the language. This is the case, for instance, of UnrealScript, JASS, and SQL. This approach has the advantages of providing a syntax close to the formal definition of the Domain-specific language, good error reporting (that in the other case is limited to that of the host language), and domain-specific optimization through code analysis. However, designing and implementing a compiler for a DSL is a hard and time-consuming task, since a compiler is a complex piece of software made of different modules that perform several translation steps. [2], for this reason this option is not always considered feasible.

Although part of a complex process, the translations steps performed by a compiler are not part of the creative aspect of designing the language [3], [5], thus they can be automated. The most common automated part is the Lexing/Parsing phase with Parser generators such as `Yacc`. A further effort in fully automating the development of a compiler has been done by employing *Meta-compilers*, that are computer programs that take as input the definition of a language (usually defined in a *meta-language*), a program written in that language and output executable code for the program. Meta-compilers usually automate not only the parsing phase, but also the type checking and the semantics implementation.

In this paper we present *Metacasanova*, a meta-compiler which meta-language is based on operational semantics [6], [7]. In Section II we further discuss how developing a compiler leads to repetitive steps that could be automated and we formulate the problem statement of this paper; in Section III we explain how the meta-language of Metacasanova is defined and what its semantics is; in Section IV we explain how the meta-compilation process is implemented in Metacasanova and how the target code is generated; in Section V we propose a further language abstraction for Metacasanova in order to improve the performance of the generated code; in Section VI we evaluate the performance of the code generated by Metacasanova after re-implementing Casanova [1] in Metacasanova, a DSL for game development, and the length of the code necessary to define the language with respect to the meta-compiler.

II. Repetitive steps in compilers development

In Section I we briefly stated that the process of developing a compiler includes several steps that are repetitive, i.e. their behaviour is always the same regardless of the language for which the compiler is built. In this section we show in what way this process is repetitive and what is the common pattern

A. Type checking

Type systems are generally expressed in the form of logical rules [4], made of a set of premises, that must be verified in order to assign to the language construct the type defined in the conclusion. For example the following rule defines the typing of an `if-then-else` statement in a functional programming language:[1]

$$\frac{\Gamma \vdash c : bool \quad \Gamma \vdash t : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{if } c \text{ then } t \text{ else } e : \tau}$$

Typing a construct of the language requires to evaluate its corresponding typing rule. In order to do so, the behaviour of each typing rule must be implemented in the host language in which the compiler is defined. Independently of the chosen language, the behaviour will always be the following

1) Evaluate a premise.
2) If the evaluation of the premise fails, then the construct fails the type check and an error is returned.
3) Repeat step 1 and 2 until all the premises have been evaluated.
4) Assign the type to the construct that is defined in the rule conclusion.

For instance, if in the example above the condition of the `if-then-else` is not a boolean expression, the type check will fail returning an error. If the condition is a boolean expression then the type τ is assigned to the `if-then-else` construct. The behaviour of the type rule above must be implemented by making use of the host language abstractions; for example in F# or CamL we would have:

```
type Expression =
...
| If of Expr * Expr * Expr

let eval (expr : Expr) =
  match expr with
  ...
  | If (condition, _then, _else) ->
      let condType = eval condition
      let thenType = eval _then
      let elseType = eval _else
      if (condType = Bool && thenType =
          elseType) then
        thenType
      else
        failwith "Error typing if statement"
```

Note that defining the behaviour in this fashion is much more verbose than the type system definition. This behaviour must be emulated for all the type rules of the language type system, but this pattern could be generalized because it requires to always perform the steps above.

[1]Note that the type rule of `if-then-else` in an imperative programming language is different.

B. Semantics

Semantics define how the language abstractions behave and can be expressed in different ways, for example with a term-rewriting system [9] or with the operational semantics [6]. For the scope of this work, we choose to rely on the operational semantics. The definition of the operational semantics of a language abstraction is, again, in the form of a logical rule where the conclusion (which is the final behaviour of the construct) is achieved if the evaluation of the premises lead to the desired results. For instance, the operational semantics of a while loop could be the following:

$$\frac{\langle c \rangle \Rightarrow \texttt{true}}{\langle \text{while } c \text{ do } L \text{ ; } k \rangle \Rightarrow \langle L \text{ ; while } c \text{ do } L \text{ ; } k \rangle}$$

$$\frac{\langle c \rangle \Rightarrow \texttt{false}}{\langle \text{while } c \text{ do } L \text{ ; } k \rangle \Rightarrow \langle k \rangle}$$

Again, the behaviour of the semantics rule must be encoded in the host language in which the compiler is being developed, but the pattern it follows is always the same. This step, depending on the implementation choice, might also require to translate this behaviour into an *intermediate language* representation that is more suitable for the next code generation phase.

C. Discussion

The examples above show how the behaviour of the type checking and semantics rules must be hard-coded in the language chosen for the compiler implementation, regardless of the fact that their pattern is constantly repeated in every rule. This pattern can be captured in a meta-language that is able to process the type system and operational semantics definition of the language and produce the code to execute the behaviour of the rules. In this work we describe the meta-language for *Metacasanova*, a meta-compiler that is able to read a program written in terms of type system/operational semantics rules defining a programming language, a program written in that language, and output executable code that mimics the behaviour of the semantics. Such a language relives the programmer from writing boiler-plate code when implementing a compiler for a (Domain-Specific) language. For this reason we formulate the following problem statement:

Problem statement: *To what extent Metacasanova eases the development speed of a compiler for a Domain-Specific Language, in terms of code length compared to the hard-coded implementation, and what is the performance of the generated code with respect to a hard-coded compiler?*

D. Related work

Fill in this section with related work, namely RML, workbenches for Haskell, and Stratego.

III. Metacasanova syntax and semantics

In the previous section we showed that the process of evaluating typing and semantics rules is always the same, regardless of the specific language implementation. We have also discussed how this evaluation must be re-implemented every time in a hard-coded compiler by using the abstractions provided by the host language, which leads to verbose code and the loss of the clarity and simplicity given by the formalisms of the type system and operational semantics. In this section we define the requirements of Metacasanova, we informally present, through an example, how a meta-program works, and we finally propose the syntax and semantics of its meta-language.

A. Requirements of Metacasanova

In order to relieve programmers of manually defining the behaviour described in Section II in the back-end of the compiler, we propose the following features for Metacasanova:

- 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.

We can see that these specifications are compatible with the definition of meta-compiler, as the software takes as input a language definition written in the meta-language, a program for that language, and outputs runnable code that mimics the code that a hard-coded compiler would output.

B. 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
```

Finally the language supports expression bindings with the following syntax:

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

C. 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>]
```

D. 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 in R2 the evaluation returns a set of results because there might be more than one rule that can successfully evaluate the premise. The following is the formal semantics of the rule evaluation in Metacasanova, based on the informal behaviour 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.

In the following section we describe how the code generation process works, namely how the `Data` types of Metacasanova are mapped in the target language, and how the rule evaluation is implemented.

IV. Code generation

In Section III we defined the syntax and semantics of Metacasanova. In this section we explain how the abstractions of the language are compiled into the generated code. We chose C# as target language because the development of Metacasanova started with the idea of expanding the DSL for game development Casanova with further functionalities. Casanova hard-coded compiler generated C# code as well because it is compatible with game engines such as Unity3D and Monogame. At the same time, C# grants decent performance without having to manually manage the memory such as for lower-level languages like C/C++. Code generation in different target languages, such as Javascript/Typescript, is possible but still an ongoing project (see Section VII).

A. Data structures code generation

The type of each data structure is generated as an interface in C#. Each data structure defined in Metacasanova is mapped to a `class` in C# that implements such interface. The class contains as many fields as the amount of arguments the data structure contains. Each field is given an automatic name `__argC` where `C` is the index of the argument in the data structure definition. The data structure symbols used in the definition might be pre-processed and replaced in order to avoid illegal characters in the C# class definition. The class contains an additional field that stores the original name of the data structure before the replacement is performed, used for its "pretty print". For example the data structure

```
Data "$i" -> int : Value
```

will be generated as

```
public interface Value {  }

public class __opDollari : Value
{
  public string __name = "$i";
  public int __arg0;

  public override string ToString()
  {
    return "(" + __name + " " + __arg0 + ")";
  }
}
```

B. Code generation for rules

Each rule contains a set of premises that in general call different functions to produce a result, and a conclusion that contains the function evaluated by the current rule and the result it produces. The code generation for the rules follows the steps below:

1) Generate a data structure for each function defined in the meta-program.
2) For each function f extract all the rules which conclusion contains f.
3) Create a `swtich` statement with a case for each rule that is able to execute the function (the function is in its conclusion).
4) In the case block of each rule, define the local variables defined in the rule.
5) Apply pattern matching to the arguments of the function contained in the conclusion of the rule. If it fails, jump immediately to the next case (rule).
6) Store the values passed to the function call into the appropriate local variables.
7) Run each premise by instantiating the class for the function used by it and copying the values into the input arguments.
8) Check if the premise outputs a result and, in the case of an explicit data structure argument, check the pattern matching. If the premise result is empty or the pattern matching fails for all the possible executions of the premise then jump to the next case.
9) Generate the result for the current rule execution.

In what follows, we use as an example the code generation for the following rule (which computes the sum of two integer expressions in a programming language):

```
eval a -> $i c
eval b -> $i d
<< c + d >> -> e
----------------
eval (a + b) -> $i e
```

From now on we will refer to an argument as *explicit data argument* when its structure appears explicitly in the conclusion or in one of the premises, as in the case of `a + b` in the example above.

1) Data structure for the function: As first step the meta-compiler generates a class for each function defined in the meta-program. This class contains one field for each argument the function accepts. It also contains a field to store the possible result of its evaluation. This field is a `struct` generated by the meta-compiler defined as follows:

```
public struct __MetaCnvResult<T> { public T
    Value; public bool HasValue; }
```

The result contains a boolean to mark if the rule actually returned a result or failed, and a value which contains the result in case of success.

For example, the function

```
Func eval -> Expr : Value
```

will be generated as

```
public class eval
{
  public Expr __arg0;
  public __MetaCnvResult<Value> __res;
  ...
}
```

2) Rule execution: The class defines a method `Run` that performs the actual code execution. The meta-compiler retrieves all the rules which conclusion contains a call to the current function, which define all the possible ways the function can be evaluated with. It then creates a `switch` structure where each `case` represents each rule that might execute that function. The result of the rule is also initialized here (the `struct` will contain a default value and the boolean flag will be set to `false`). Each `case` defines a set of local variables, that are the variables used within the scope of that rule.

3) Local variables definitions and pattern matching of the conclusion: At the beginning of each `case`, the meta-compiler defines the local variables initialized with their respective default values. It also generates then the code necessary for the pattern-matching of the conclusion arguments. Since variables always pass the pattern-matching, the code is generated only for arguments explicitly defining a data structure (see the examples about arithmetic operators in Section III) and literals. If the pattern matching fails then the execution jumps to the next `case` (rule). For instance, the code for the following conclusion

```
...
-------------
eval (a + b) -> $i e
```

is generated as follows

```
case 0:
{
  Expr a = default(Expr);
```

```
  Expr b = default(Expr);
  int c = default(int);
  int d = default(int);
  int e = default(int);
  if (!(__arg0 is __opPlus)) goto case 1;
  ...
}
```

Note that an explicit data argument, such in the example above, might contain other nested explicit data arguments, so the pattern-matching is recursively performed on the data structure arguments themselves.

4) Copying the input values into the local variables: When each function is called by a premise, the local values are stored into the class fields of the function defined in Section IV-B1. These values must be copied to the local variables defined in the `case` block representing the rule. Particular care must be taken when one argument is an explicit data. In that case, we must copy, one by one, the content of the data into the local variables bound in the pattern matching. For example, in the rule above, we must separately copy the content of the first and second parameter of the explicit data argument into the local variables `a` and `b`. The generated code for this step, applied to the example above, will be:

```
__opPlus __tmp0 = (__opPlus)__arg0;
a = __tmp0.__arg0;
b = __tmp0.__arg1;
```

Note that the type conversion from the polymorphic type `Expr` into `__opPlus` is now safe because we have already checked during the pattern matching that we actually have `__opPlus`.

5) Generation of premises: Before evaluating each premise, we must instantiate the class for the function that they are invoking. The input arguments of the function call must be copied into the fields of the instantiated object. If one of the arguments is an explicit data argument, then it must be instantiated and then its arguments should be initialized, and then the whole data argument must be assigned to the respective function object field. After this step, it is possible to invoke the `Run` method of the function to start its execution. The first premise of the example above then becomes (the generation of the second is analogous):

```
eval a -> $i c
```

```
eval __tmp1 = new eval();
__tmp1.__arg0 = a;
__tmp1.Run();
```

6) Checking the premise result: After the execution of the function called by a premise, we must check if a rule was able to correctly evaluate it. In order to do so, we must check that the result field of the function object contains a value, and if not the rule fails and we jump to the next case (rule), which is performed in the following way:

```
if (!(__tmp1.__res.HasValue)) goto case 1;
```

If the premise was successfully evaluated by one rule, then we must check the structure of the result, which leads to the following three situations:

1) The result is bound to a variable.
2) The result is constrained to be a literal.
3) The result is an explicit data argument.

In the first case, as already explained above, the pattern matching always succeeds, so no check is needed. In the second case, it is enough to check the value of the literal. In the last case, all the arguments of the data argument must be checked to see if they match the expected result. In general this process is recursive, as the arguments could be themselves other explicit data arguments. If the result passes the check, then the result is copied into the local variables, in a fashion similar to the one performed for the function premise. For instance, for the premise

```
if (!(__tmp1.__res.HasValue)) goto case 1;
```

the meta-compiler generates the following code to check the result

```
if (!(__tmp1.__res.Value is __opDollari)) goto
     case 1;
__MetaCnvResult<Value> __tmp2 = __tmp1.__res;
__opDollari __tmp3 = (__opDollari)__tmp2.Value
    ;
c = __tmp3.__arg0;
```

7) Generation of the result: When all premises correctly output the expected result, the rule can output the final result. In order to do that, the generated code must copy the right part of the conclusion (the result) into the `__res` variable of the function class. If the right part of the conclusion is, again, an explicit data argument, then the data object must first be instantiated and then copied into the result. For example the result of the rule above is generated as follows:

```
res = c / d;
__opDollari __tmp7 = new __opDollari();
__tmp7.__arg0 = res;
__res.HasValue = true;
__res.Value = __tmp7;
break;
```

After this step, the rule evaluation successfully returns a result.

V. Compile-time inlining with Functors

VI. Evaluation

VII. Conclusion and future work

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 *Joint International Conference on Serious Games*, pages 123–134. Springer, 2015.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986.

[3] Erwin Book, Dewey Val Shorre, and Steven J Sherman. The cwic/36o system, a compiler for writing and implementing compilers. *ACM SIGPLAN Notices*, 5(6):11–29, 1970.

[4] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.
[5] Krzysztof Czarnecki, Ulrich W Eisenecker, G Goos, J Hartmanis, and J van Leeuwen. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, 15, 2000.
[6] Plotkin G.D. A structural approach to operational semantics. Technical report, Computer science department, Aarhus University, 1981.
[7] Gilles Kahn. Natural semantics. *STACS 87*, pages 22–39, 1987.
[8] Samuel N Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14:149–168, 1998.
[9] Jan Willem Klop et al. Term rewriting systems. *Handbook of logic in computer science*, 2:1–116, 1992.
[10] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
[11] Anthony M Sloane. Post-design domain-specific language embedding: A case study in the software engineering domain. In *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, pages 3647–3655. IEEE, 2002.
[12] Arie Van Deursen, Paul Klint, Joost Visser, et al. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.
[13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.