Institute of Computer Science
University of Wrocław

Syntax-extending and type-reflecting macros in an object-oriented language

Master Thesis
Wrocław 2005

Kamil Skalski

Contents

1 Preface

We [1] present a meta-programming system embedded into Nemerle language. It provides convenient tool for extending language, generating code, partial evaluation and code transformation.

The paper is organized as follows. Section 2 introduces the notion of meta-programming system and presents the core features of our design. In section 3 we present its implementation in Nemerle compiler. Section 4 presents application of our system's capabilities to partial evaluation. Then we extend considered language subset with object-oriented constructs in section 5 and present how meta-programming applies to concepts natural to languages like Java and C#. In sections 6 and 7 additional features like syntax extensions and type reflection are discussed. Finally, section 8 summarizes the related work on the topic of meta-programming.

2 Introduction

Meta-programming is a powerful technique for software development. It was incorporated into several languages, like Lisp macros [31], Scheme hygienic macros [1], C [28] preprocessor-based macros, C++ [12] template system and finally ML languages like MetaML [36] and Haskell Template Meta-programming [30]. It is a convenient way of improving expressibility, performance, code reuse and programmer's productivity.

Despite their baroque nature C++ templates are used in fascinating ways that extend beyond the wildest dreams of the language designers [3]. Also the wide use of C preprocessor [15] shows that users find it convenient and ignore its error prone semantics (purely textual macros). But drawbacks of these two designs influenced the recent proposals. Besides all its advantages, meta-programming has not been incorporated into today's mainstream languages like Java and C#. Our proposal is aimed to fill the gap between the extremly expressive Scheme macros and popular object-oriented languages, which lack the convenient macro system.

The essence of *meta-programing* is automated program generation. During this process programs are treated as *object programs*, which becomes the data supplied to *meta-programs*. They can be then arbitrarily transformed or analyzed and the final result is compiled just like a regular program. These operations may be repeated or take place in stages. In the latter case the generated programs can generate other programs and so on. A

[1] In this thesis, I use "we" to refer primaly to myself, though many of the design decisions were made with other developers and after discussion within the community.

meta-program may construct object-programs, combine their fragments into larger object-programs, observe their structure and other properties, decompose them. Meta-programs include things like compilers, interpreters, type checkers, theorem provers, program generators, transformation systems. In each of these a program (the meta-program) manipulates a data-object representing a sentence in a formal language (the object-program).

The main real life applications of meta-programming are:

- adding new constructs to the language, which are not expressible in core language as simple functions calls,
- generating programs specialized to some task,
- reducing the need of repetitive writing of similar code patterns,
- partial evaluation
- other optimization techniques based on compiler's static information about program.

2.1 Taxonomy of meta-programs

Following a simple taxonomy presented in [29] we can describe Nemerle as:

- Both program generation and analysis system. It is able to perform algorithmic construction of programs and decomposing them using pattern matching.
- Compile-time program generator, which is executed by compiler and whose results are in-lined into the compiled program.
- Manually annotated. The body of program generator is divided into meta-program and object-program. Staging annotations are used to separate pieces of the program.
- Homogeneous. The meta-language and object-language are the same, which simplifies both usage (user do not need to learn two languages) and scaling the system to multilevel and run-time code generator.
- Two level language. Object-language cannot express a meta-program. This limitation is valid for compile-time meta-system, because there is rarely need for multi-stage programs.

- Quasi-quote representation of object-programs. It gives the same ease of use as string representation and algebraic-types' assurance about syntactic correctness of generated code.
- Object-programs are dynamically typed. In contrast to statically typed meta-languages, which gives assurances about type-correctness of generated code, we type-check programs only after they are generated. This is more flexible (allows imperative generation of programs) and is still a viable design in compile-time system - programs are still type-checked during the compilation process.

The more detailed list of features is presented in section 2.3.

2.2 Subset of Nemerle language

Nemerle is a general purpose object-oriented programming language with features of ML-like functional languages (that is type inference, algebraic data-types, pattern matching and first class functional values). For the purpose of this section we define the subset of Nemerle's expressions. The object-oriented elements are described in section 5.

Here are the BNF rules for core Nemerle expressions.

⟨*program*⟩ ::= ⟨*expr*⟩+

⟨*expr*⟩ ::=
  ⟨*var*⟩
| ⟨*literal*⟩
| `def` ⟨*var*⟩ `=` ⟨*expr*⟩
| `def` ⟨*var*⟩ `(` ⟨*var*⟩ (`,` ⟨*var*⟩)* `)` ⟨*block*⟩
| ⟨*expr*⟩ `(` [⟨*expr*⟩ (`,` ⟨*expr*⟩)*] `)`
| ⟨*block*⟩

⟨*block*⟩ ::= `{` ⟨*expr*⟩ (`;` ⟨*expr*⟩)* `}`

⟨*var*⟩ ::= set of identifiers

⟨*literal*⟩ ::= set of literals (integers, strings, etc.)

The `def` keyword has similar meaning to OCaml `let`. For example

```
def x = 5; x
```

is equivalent to

```
let x = 5 in x
```

and

```
def f (x) { x + 1 }
```

relates to

```
let f x = x + 1
```

2.2.1 Meta-programming extension

Meta-language is a language for programming operations on object language. It usually has its own syntax for describing various constructs of the *object language*. For example, in our system, `<[ 1 + f (2 * x) ]>` denotes the syntax tree of expression `1 + f (2 * x)`. This idea is called *quasi-quotation*. The prefix *quasi* comes from the possibility of inserting values of meta-language expressions into the quoted context – if `g(y)` is such an expression, we can write `<[ 1 + $(g (y)) ]>`, which describes a syntax tree, whose second part is replaced by the result of evaluation of `g(y)`.

We can now consider extending the language with meta-programming constructs. Formally, we add following two productions to the core expressions:

⟨*expr*⟩ ::=
  | `<[` ⟨*expr*⟩ `]>`
  | `$` ⟨*expr*⟩

Meta-language allows us to operate on object programs easily. But at some point of time we want to merge them into the final program and finally run them. Most general meta-programming systems provide *eval* or *run* construct for executing generated code. This implies that object code is interpreted during run-time. In Nemerle we have mainly focused on compile-time meta-programming, which yields that at run-time the whole program is already compiled and won't change.

Our model assumes that meta-programs are run by compiler and this way all the code generation and transformation is performed at compile-time. Here, the role of *run* construct is taken over by *macros*. Macros are meta-programs, which are executed by compiler for every occurrence of their invocation in program.

Macro is defined by the following rule:

⟨*macro definition*⟩ ::= `macro` ⟨*var*⟩ `(` [⟨*var*⟩ (`,` ⟨*var*⟩)*] `)` ⟨*block*⟩

and the *program* production is extended with

⟨*program*⟩ ::=
| ⟨*macro definition*⟩ ⟨*program*⟩

Macro invocation has the same syntax as a standard function call. For example `f(x)` is a function call, but if the name of functional value `f` binds to the visible macro, like `macro f (e) { ... }`, then instead of generating run-time function call the macro is executed and the generated code is inlined and processed further.

2.3 Features

Meta-programming systems varies on the design and implemented features. In this section we summarize most components of our system and give the basic reasoning behind our design.

2.3.1 Syntactic correctness

There are two common ways of representing object programs - as strings (like in C preprocessing macros and most scripting languages like Bash, Perl, Python) or as data-types (advanced meta-programming systems). With the string encoding, we represent the code fragment `f (x)` simply as `"f (x)"`. This representation has the advantage of straightforward usage, programs can be constructed and concatenated by simple operations on strings. But deconstruction of programs and reasoning about their syntactic correctness becomes quite hard. For example `"f (x("` is a valid string literal, but it does not represent syntactically valid object program.

When programs are represented using ML-like data-types (or any data structures specially crafted for this purpose), their syntactic correctness is implied by type system of meta-language (or invariants ensured by used data-structure). For example using data-types in Nemerle (called variants there) we can represent expressions of simple lambda calculus language defined with the BNF rule

⟨*expr*⟩ ::=
⟨*x*⟩
| ⟨*expr*⟩ ⟨*expr*⟩
| λ ⟨*x*⟩ . ⟨*expr*⟩

⟨*x*⟩ ::= set of identifiers

by the following variant

```
variant Expr {
  | Variable { name : string }
  | Apply { func : Expr; parm : Expr }
  | Lambda { var : string; body : Expr }
}
```

The data-type encoding is roughly equivalent to abstract syntax trees of the lambda calculus language. Every object of type `Expr` corresponds to the valid lambda expression and vice-verse. For example code fragment $\lambda x.x$ is created with `Lambda ("x", Variable ("x"))`. This approach have also one more great advantage, that object code can be inspected using the standard Nemerle pattern matching facility.

Our system uses similar data-type representation internally, but we also provide a special syntax called code quotations for constructing object code. It allows us to specify the same code fragment as `<[` $\lambda x.x$ `]>` This way we combine the ease of use of string encoding with the strength of the data-type encoding. Quoted expression is accepted by parser only if it is syntactically correct and Nemerle type system will ensure that all created objects are correct with regard to variant defining expressions.

2.3.2 Hygiene

Hygiene relates to the problem with names capture in Lisp macros, resolved later in Scheme. It specifies that variables introduced by a macro may not bind to the variables used in the code passed to this macro. Particularly variables with the same names, but coming from different contexts, should be automatically distinguished and renamed.

Consider the following example:

```
macro identity (e) { <[ def f (x) { x }; f ($e) ]> }
```

Calling it with `identity (f(1))` might generate a confusing code like

```
def f (x) { x }; f (f (1))
```

To prevent names capture, all macro-generated variables should be renamed to their unique counterparts, like in

```
def f_42 (x_43) { x_43 }; f_42 (f (1))
```

In Nemerle we achieve hygiene by automatically associating contexts to each variable created using `<[ ]>` quotations. Details are presented in section 3.4.

Breaking hygiene The automatic hygiene in Nemerle can be overridden in a controlled way. It is possible to create variables, which bind to names from context of macro use site.

This way we can generate code, which introduces new names to the place where the macro has been invoked. Also it is useful in embedding domain-specific languages, which reference symbols from the original program.

```
macro interpret (symbol : string) {
  <[ $(symbol : usesite) + 1 ]>
}

def x = 4; interpret ("x")
```

Breaking of hygiene is necessary here, because symbols are created by parser of embedded language and we generate code, which need to have the same context as variables from invocation place of macro.

Unhygienic variables Sometimes it is useful to completely break hygiene, where programmer only want to experiment with new ideas. From our experience, it is sometimes hard to reason about correct contexts for variables, especially when writing class level macros (section 5.3). In this case it is useful to be able to easily break hygiene and after its basic functionality is tested programmer can fix the hygiene issues.

Nemerle provides it with `<[ $("id" : dyn) ]>` construct. It makes produced variable break hygiene rules and always bind to the nearest definition with the same name.

2.3.3 Lexical scoping

In Nemerle variables and global symbols are statically scoped. It means that the place where they can be referenced is defined by the text of the program. For local variables it is the enclosing block or function, for global symbols the set of imported modules and namespaces.

With object programs, things are getting more subtle. We must first define how we understand lexical scoping of symbols in object code. In opposition to strong statically typed meta-programming languages like MetaML [36] we do not require names in object-programs to follow their static scoping in meta-program. Their scope is checked with respect to the final structure of generated program.

The following code fragment

```
def body = <[ x ]>;
def function = <[ def f (x) $body ]>;
<[ $function; f (1) ]>
```

is a valid Nemerle program, but its counterpart would be rejected by MetaML. MetaML would try to bind `x` in `<[ x ]>` to some definition in scope, but in Nemerle we do not statically type-check object programs. It is done only for final program, which in this case would be

```
<[ def f (x) { x }; f (1) ]>
```

This design gives much better expressibility, but we loose guarantees about type-safety of generated object-programs. The full discussion of these two approaches can be found in section 8.1.1.

Hygiene Still, as described in previous section, we maintain locality of symbols in generated code. System automatically preserves us from inadvertent capture of names from external code and from the place of macro's usage.

This combination of ideas has proved to be very useful and seems to give a good balance between expressibility of Lisp dynamic macros and hygienic systems like Template Haskell and MetaML.

Global symbols

Definition 1 (Global environment) *It is a pair (C, O) where C is the current namespace in which given class resides and O is the set of imported namespaces, opened classes (the* `using Name.Space;` *construct) and defined type aliases. The notion of* namespace *is extensively used in* `.NET` *for modularization of programs and organizing large libraries.*

We decided that object-programs should carry global environment of the meta-program they are generated in. This is a logical choice, since macro can be executed in many different contexts and we expect references to external modules and classes to always bind to the same entities.

For example, consider following definition of macro:

```
using System.Console;

macro log (content) {
  <[
    WriteLine ("{0}: {1}", content.Location.File, content)
  ]>
}
```

where `WriteLine` is a function from `System.Console` module. This macro will work no matter what modules are open in the place of its use, because quoted `WriteLine` reference carries its global environment from meta-program.

2.3.4 Syntax extensions

One of the main purposes of meta-programming system is language extensibility. Macros allow us creating complex constructs, which were not expressible otherwise with standard built-in mechanisms, like calling functions from predefined library.

For example consider addition of C-like **while** loop. We can define it using macro producing recursive local function and its call.

```
macro while_macro (condition, body)
{
  <[
    def while_loop () {
      when ($condition) {
        $body;
        while_loop ()
      }
    }
    while_loop ()
  ]>
}
```

We can then use it as `while_macro (true, print ("Hello"))`. But it still lacks special syntax of original C construct. We can add it using Nemerle syntax extending capabilities. We extend definition of our macro to

```
macro while_macro (condition, body)
syntax ("while", "(", condition, ")", body)
{ ... }
```

and now it is possible to write `while (true) print ("Hello")`. This feature simplifies greatly a compiler implementation, but also gives user a powerful tool for customizing language to her needs.

Syntax extending rules require that their first element is a keyword or operator. Defining such a macro adds new keywords and parsing rules to the namespace where it resides. When user do not import such a namespace, no changes are visible. This way we can safely extend language not interfering with existing code.

Detailed description of accepted syntax rules and its implementation is described in section 6.

2.3.5 Type reflection

Macros are meant to extend compiler in a scalable way. In order to do this it is often necessary that they have relatively the same capabilities to analyze

given program as the compiler itself. In statically typed language that means querying about the types of program fragments, inheritance relation, symbols exported from external modules, etc.

In Nemerle we give access to the compiler API inside macros. This allows meta-programmer to query about various information dependent on compilation context (loaded libraries, passed command line options), but also on current compilation state (sub-typing information about analysed classes in program, visible local variables in given place, etc.). One of the hardest tasks is reflecting the type of given expression, especially in presence of type inference.

Consider an example macro **foreach**:

```
macro foreach (var, collection, body)
{ ... choose implementation according to collection type ... }

def a = array [1, 2, 3];
def l = List ();
l.Add (1);
foreach (x, a, ...);
foreach (x, l, ...);
```

It is necessary to use different algorithm for iterating over an array or a list and macro should be able to choose it. In order to do this it must query the type of `collection` expression. The straightforward way would be to just execute typing function on it inside macro, but it becomes a problem when current compilation state do not supply needed information to infer the type. In fact Nemerle type inference is based on deferred type constraints solving and often the actual type of expression is known only after typing expressions proceeding it in program text.

Standard approach to macros fails on this issue, because it treats them purely as parse-tree to parse-tree transformations, which are expanded before typing is performed. We address this issue by deferring macro expansion until type information requested by user is known. The algorithm is described in section 7.

3 Implementation of core system

In this section we present the details of Nemerle macros' core features implementation. This description focuses only on expression based macros, but also gives the insight into presented system, which will be extended in following sections.

Macro system in essence is mainly an extension of compiler, thus it is closely bundled with it. For this reason we have to describe its implementation in context of internal compiler architecture. Conversely, the abstract description of meta-programming part of Nemerle does not necessarily need to get into implementation details. In fact, the goal of most programming language designs is to hide underlying representation of language constructs minimizing loss of expressibility and generality at the same time. In this section we focus on internal workings of compiler and how we have built our system to meet its basic requirements.

We start with description of involved compiler passes, interaction of macro execution with them and then how we obtained the two core features - hygiene and lexical scoping of symbols in object code.

3.1 Compiler passes

When we analyse how compilers normally work we learn that it is a pipeline of very basic operations.

- Loading external libraries - here we load and initialize macros defined in all referenced modules
- Scanning of textual program data into a stream of tokens. This is the standard technique incorporated into roughly all compilers.
- Preparsing - the initial processing of lexer tokens, by grouping them into tree of parentheses. This is not the common stage in most designs. We use it mainly for deferring parsing of program parts in Nemerle syntax extensions described in section 6.
- Main parsing phase. It builds the syntax trees, which are the main data-structure macros are operating on. Parser must take syntax extension rules into account when analysing preparsed token tree.
- Class hierarchy building is responsible for creating inheritance graph of classes defined in the program and binding types of class members (fields, methods). This stage is important, because it involves expanding of top level macros (section 5) and building of classes representing macros.
- Expression typing is the crucial stage for our implementation of meta-system. It involves translation of code quotations and macro expansion (including delayed expansions described in section section 7.2).

- Code generation is the final compilation phase, where `.NET` meta-data and Intermediate Language is created. It is orthogonal to meta-system.

3.2 Macros

Definition 2 (Program fragment) *We will call the node of parse tree / abstract syntax tree a program fragment. It is the representation that compiler uses for untyped fragments of program.*

Formally it is a term of sort `PExpr` *following the definition of Nemerle language subset in section 2.2:*

⟨*PExpr*⟩ ::=
    Var
  | `Literal (` *Literal* `)`
  | `Def (` *Var* `,` *PExpr* `)`
  | `DefFun (` *Var* `,` *list [Var]* `,` *PExpr* `)`
  | `Call (` *PExpr* `,` *list [PExpr]* `)`
  | `Block (` *list [PExpr]* `)`

⟨*Var*⟩ ::= `Ref (` *string* `)`

⟨*Literal*⟩ ::= *node representing literal (integers, strings, etc.)*

Definition 3 (Macro) *is an instance of class implementing a special interface* IMacro *with* $Run : list[PExpr] \rightarrow PExpr$ *method. It transforms list of program fragments into the new program fragment.*

Macro class is automatically generated by compiler basing on macro declaration (as defined in section 2.2.1).

A key element of our system is the execution of meta-programs during the compile-time. To do this they must have an executable form and be compiled before they are used.

Macros after compilation are stored in assemblies (compiled libraries of code). All macros defined within an assembly are listed in its meta-data. Therefore during the compilation, when linking of an assembly is requested by user, we can construct instances of all macro classes and register them by names within the compiler.

For this purpose we define a $LookupMacro : PExpr \rightarrow IMacro$ function for accessing macro instances by name. It accepts only `Ref` nodes of program fragments and is used during macro expansion.

3.2.1 Separate Compilation

The current implementation requires macros to be compiled separately, before the compilation of the program that uses them. That is, all the macros are loaded in the first phase of the compilation and the *LookupMacro* function is then fully defined.

This results in inability to define and use a given macro in the same compilation unit. While we are still researching the field of generating and running macros during the same compilation our current approach also has some advantages.

The most important one is that it is simple and easy to understand – one needs first to compile the macros (probably being integrated into some library), and then load them into the compiler and finally use them. This way the stages of compilation are clearly separated in a well understood fashion – an important advantage in the industrial environment where metaprogramming is a new and still somewhat obscure topic.

The main problem with ad-hoc macros (introduced and used during the same compilation) is that we need to first compile transitive closure of types (classes with methods) used by given macro. This very macro of course cannot be used in these types.

This issue may be hard to understand by programmers ("why doesn't my program compile after I added new field to this class?!"). Moreover, such a dependency-closed set of types and macros can be easily split out of the main program into the library.

Experiences of the Scheme community show [16] how many problems arise with systems that do not provide clear separation of compilation stages. Indeed, to avoid them in large programs, manual annotations describing dependencies between macro libraries are introduced.

3.2.2 Macro expansion

Macro expansion is a process of substituting every occurrence of macro invocation in program with the value of its evaluation.

Formally it is a process of replacing occurrences of

$$\texttt{Call}(t, parameters)$$

in program fragment where t is `Ref` containing the name of a visible macro m, with the result of $m.Run$'s execution supplied with the list of program fragments contained in $parameters$ list. Following inference rule specifies this step:

$$\frac{LookupMacro(t) \rightarrow m \quad m.Run(parameters) \Rightarrow^{run} e \quad e \rightarrow e'}{Call(t, parameters) \rightarrow e'} (Expand)$$

The process is recursive, that is a single macro expansion can yield another expansions, if there are occurrences of macro invocation in substituting program fragment e.

3.3 Quotations

The quasi-quotes was first incorporated into Lisp programming language. Its early use is described in [32]. The Lisp back-quote begins a quasi-quotation, and a comma preceding a variable or a parenthesized expression acts as an anti-quotation indicating that the expression should evaluate to object-code. A short history of *Quasiquotation in LISP* [6].

The quotation system in essence is a syntactic shortcut for explicitly constructing syntax trees using algebraic data-type representation of object code. For example `f(x)` expression is internally represented by

```
Call (Ref ("f"), [Ref ("x")])
```

which is equivalent to `<[ f (x) ]>`. Translating the quotation involves "lifting" the syntax tree by one more level – we are given an expression representing a program (its syntax tree) and we must create a representation of such expression (a larger syntax tree). This implies building a syntax tree of the given syntax tree, like

```
Call (Ref ("f"), [Ref ("x"))]
=>
Call (Ref ("Call"),
      [Call (Ref ("Ref"), [Literal (StringLiteral ("f"))]),
       Call (Ref ("Cons"),
             [Call (Ref ("Ref"),
                    [Literal (StringLiteral ("x"))]),
                     Call (Ref ("Nil"), [])])])
```

or using the quotation

```
<[ f (x) ]>
=>
<[ Call (Ref ("f"), [Ref ("x")]) ]>
```

Now splicing means just "do not lift", because we want to pass the value of the meta-language expression as the object code. Of course it is only valid when such an expression describes (is type of) the syntax tree.

3.4 Hygiene

In this section we describe how we achieve hygiene in our system. As said before, we associate "context" or "color" with every identifier in the program. The *Var* from definition 2 must be updated to

⟨*Var*⟩ ::= `Ref ( string , int ,` global environment `)`

Now identifiers from the object program are denoted $Ref(v, c, g)$ where v is the name of identifier, c is its color, and g is global environment for a given symbol.

The point is to assign distinct color to symbols generated by each macro invocation. If all the identifiers originating from a single macro execution use the same context, the code returned from it preserves correct binding as expected when generating code. Neither external declarations can capture symbols from macro nor generated declarations are visible outside.

Our coloring system is quite simple. All plain identifiers introduced in quotation receive the color of the current macro invocation (which is unique for every single macro expansion). The identifiers marked with splicing `$(id : usesite)` receive the color of the code that called the macro. This can be the color of the initial code obtained from parsing a program, as well as the color of some intermediate macro invocation. The example of such a process is shown in figure 1.

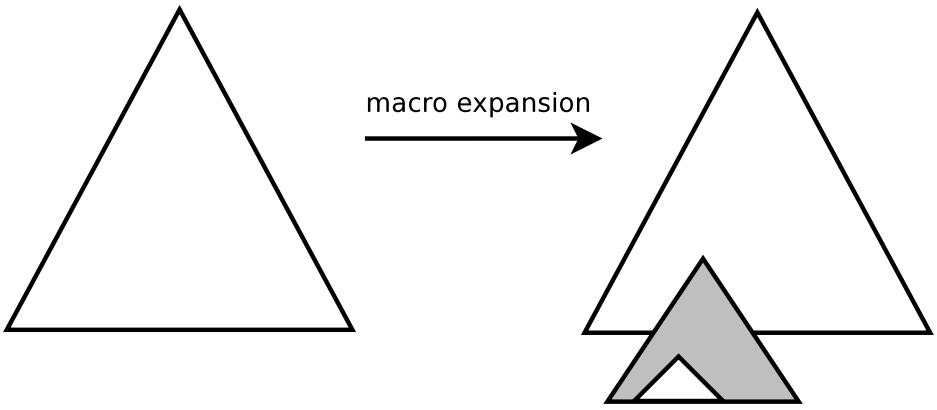

Figure 1: Macro expansion. The gray triangle is a parsetree generated by macro, where symbols have unique color assigned. The little white triangle inside has two possible origins - it is a parameter of macro spliced into its result or parsetree generated using *usesite* symbols.

After all macros are expanded and colors resolved we can say what each name binds to. Regular lexical scoping rules apply – some terms define

symbols, some other use it. Use refers to the nearest preceding declaration with the same color. If there is no such declaration – the symbol is looked up in global environment enclosed in each symbol.

3.4.1 Formal rewriting rules

The algorithm described above can be written using a set of rewrite rules operating on `PExpr` terms representing program fragments. It is executed before expression typing phase and will expand all macro invocations and assign proper color and global environment to every identifier in program.

Identifiers introduced by macros (inside quotations) in a hygienic way are denoted $Ref(v, current(), g)$ while identifiers introduced with splicing `$(x : usesite)` construct are marked $Ref(v, usesite(), g)$. The global environment g comes from the context within which the macro was defined. Top-level object code is already colored with a single unique color and proper environment, so identifiers take form $Ref(v, c, g)$ where $c \notin \{current(), usesite()\}$.

The $LookupMacro : PExpr \to IMacro * int * global\ environment$ function is extended after the Ref node and now returns a tuple of macro instance, color and context contained inside the supplied Ref.

$$\frac{e_1 \to e_1' \ \ldots \ e_n \to e_n}{\mathcal{F}(e_1, \ldots, e_n) \to \mathcal{F}(e_1', \ldots, e_n')}\,(Mon) \quad \text{where } \mathcal{F} \notin \{Ref, Call\}$$

$$\frac{}{Ref(v, x, g) \to Ref(v, x, g)}\,(MonVar)$$

$$\frac{LookupMacro(t) \to (m, u, g') \quad m.Run(ps) \Rightarrow^{run} e \quad e \Rightarrow_c^{(u,g')} e' \quad e' \to e''}{Call(t, ps) \to e''}\,(Expand)^2$$

$$\frac{LookupMacro(t) \not\to (m, u, g') \quad t \to t' \quad ps \to ps'}{Call(t, ps) \to Call(t', ps')}\,(MonCall)$$

$$\frac{e_1 \Rightarrow_c^{(u,g')} e_1' \ \ldots \ e_n \Rightarrow_c^{(u,g')} e_n}{\mathcal{F}(e_1, \ldots, e_n) \Rightarrow_c^{(u,g')} \mathcal{F}(e_1', \ldots, e_n')}\,(ColMon) \quad \text{where } \mathcal{F} \notin \{Ref\}$$

$$\frac{}{Ref(v, usesite(), g) \Rightarrow_c^{(u,g')} Ref(v, u, g')}\,(ColUse)$$

$$\frac{}{Ref(v, current(), g) \Rightarrow_c^{(u,g')} Ref(v, c, g)}\,(ColCur)$$

$$\frac{}{Ref(v, x, g) \Rightarrow_c^{(u,g')} Ref(v, x, g)}\,(ColSkip) \text{ where } x \notin \{current(), usesite()\}$$

[2] where c is a fresh color

Definition 4 *We say that e is* valid *if it does not contain terms of the form* $Ref(v, current(), g)$ *and* $Ref(v, usesite(), g)$.

Definition 5 *We say that e is in* normal form *if it is valid and does not contain any subterm* $\texttt{Call}(t, ps)$ *where* $LookupMacro(t) \rightarrow (m, u, g)$.

Normal form is thus an object code without macro invocation and with full color information.

Theorem 1 *For any valid e, there exists its* e' *in normal form, such that it can be proved that* $e \rightarrow e'$.

Proof 1 *Rules for both* $\rightarrow$ *and* $\Rightarrow$ *are syntax-directed and defined for all terms. Thus, for any e there exists* e'*, such that* $e \rightarrow e'$*. Moreover, usage of* $\rightarrow$ *eliminates all occurrences of* $\texttt{Call}(t, ps)$ *where* $LookupMacro(t) \rightarrow (m, u, g)$*, usage of* $\Rightarrow$ *guarantees elimination of all* $current()$ *and* $usesite()$ *introduced by macros.*

Note that this process does not necessarily terminate (because of *Run* stage), but this is the very essence of our design. Macros are programs created by user and it is her responsibility to make it terminating. What can be proved is that if all macros terminate for all possible inputs, then entire macro expansion process terminates.

3.5 Lexical scoping

The coloring algorithm yields proper lexical scoping of local variable declarations in generated code. But also references to globally visible symbols are propagated correctly from macro declaration. The $ColUse$ rule introduces color and context from macro's invocation site into symbol and $ColCur$ rule stores context originating from the macro declaration site. This way global references are interpreted the same way in quoted code and inside macro that generates it.

3.5.1 Cross stage persistence of global symbols

Some meta-programming languages (like MetaML and MetaOCaml) provide the feature of cross stage persistence.

Definition 6 (Cross stage persistence) *Symbol is cross stage persistent, when it is bound at one level and it is used at a higher level.*

In Nemerle we allow cross stage persistence only of global symbols, which is its very special case. Here the "bound at one level" means that symbol is imported and visible at that level, so the only difficulty in using it at higher level is proper names translation, which is achieved by our algorithm.

In Nemerle all the other objects must be explicitly lifted and are not cross-stage persistent. We have found this limitation a consistent choice in connection with the fact that symbol references are resolved *after* code is generated, not earlier. Consider

```
def e = <[ def x = 5 ]>;
def x = 10;
<[ $e1; x ]>
```

In statically typed meta-language the reference to `x` in last line would bind to the declaration in second line (and the `x` value would need to be cross stage persistent). If that declaration were not present, the error would occur. In Nemerle meta-variables are not considered directly visible in object code and `x` is bound in a standard way to the declaration contained in `e`.

On the other hand, a general cross stage persistence gives a very powerful and convenient tool for utilizing values computed at current stage in generated code. In Nemerle we provide a set of *Lift* functions, which perform the translation of some values into code recreating them (it is easy only for constant literals and built-in types, e.g. lists of literals). For other types user need to create them by her own. It would be possible to extend Nemerle with a new construct like `$(x : lift)` which provided a real cross stage persistence of `x`, but it would be inevitably limited. The implementation would need to create a special storage for such values in the meta-program's binary, serialize them somehow, store in that storage and generate a code, which would call the deserialization routine. The limitation comes from our compilation model - we compile meta-program to a separate binary and then use it as code transformation function when compiling other programs. Meta-programs and object-programs never execute side-by-side. The hypothetical *Run* function would remove this limitation, as meta-programs were executed at run-time.

4 Partial evaluation

Partial evaluation [23] is a program optimization technique also known as *program specialization*. It is based on the same concept as projection in analysis or currying in logic, which in essence, is specializing two argument function to one argument function by fixing one input to a particular value.

In programming languages setting it requires program transformation, which can be performed conveniently using meta-programming.

For a given program $P(s, d)$ with inputs s and d we generate a new program $P_{s_1}(d)$ by fixing s to s_1.

$$(P(s, d), s_1) \rightarrow P_{s_1}(d)$$

such that

$$\forall s, d \quad P(s, d) = P_s(d)$$

In general purpose partial-evaluation systems, the generation of P_{s_1} is performed by *partial evaluator* based on static input s_1 and program P (usually as text). It is usually done automatically: first, during so called *binding time analysis* [22] step partial evaluator determines, which parts of program are static and which parts are dynamic, then the specialized program can be generated. The most common problem with such an automatic process is that it's inaccurate, requires very complex analysis of code and is almost out of control. The alternative approach is based on manual annotating program parts to be computed statically or dynamically. The concept of multi-stage programming [35] takes it even further and allow specifying several stages of computation. This way programmer has great control over the partial evaluation.

Our system provides a good framework for code generation and can be used as a basis for automatic partial evaluator, though in this thesis we show its capabilities as a two-stage programming system. The interesing work on merging those two approaches exists [26] and is a good research topic for future Nemerle development.

The following sections provide examples, which are both presentation of our system in practice and motivation for it in context of partial evaluation.

4.1 Power function - classical example

The very first example of using meta-programming as partial evaluation tool is specializing a power function $pow : R \times N \rightarrow R$

$$pow(x, 0) = 1.0$$
$$pow(x, n) = x * pow(x, n - 1) \quad \forall n > 0$$

One of the most efficient ways of computing it is the logarithmic power algorithm. Its implementation in Nemerle is presented below:

```
def sqr (x) { x * x }

def power (x, n)
{
  if (n == 0)
    1.0
  else
    if (n % 2 == 0) // even
      sqr (power (x, n / 2))
    else
      x * power (x, n - 1)
}
```

As we can see it divides n by 2 if it is even and decrements by 1 if it is odd. In former case it performs square operation and in latter multiplication by x parameter. Note that the pattern of those operations depends only on n - parameters x and n are independent.

Here we come to the point - when we want to have specialized power function for some n, then we know exactly what operations it will perform on second argument. For example optimal pattern for x^5 is $x * x^{2^2}$. Now we want the compiler to be able to generate this optimal set of operations for us. Here is the macro to perform this operation:

```
macro power1 (x, n : int)
{
  def pow (n) {
    if (n == 0)
      <[ 1.0 ]>
    else
      if (n % 2 == 0) // even
        <[ sqr ($(pow (n / 2))) ]>
      else
        <[ $x * $(pow (n - 1)) ]>
  }
  pow (n);
}
```

We will give two different, but in general equivalent descriptions of what is happening here.

First you can view this as staged computation [34]. We defer performing of `sqr` and `*` operations until `x` is known at run-time, but we are free to perform all the others like comparisons of `n` and recursion at earlier stage. In the result only the optimal pattern of multiplication and square operations will be executed at run-time. The `power1` macro is just a function, which performs first stage of computation (at compile time). The result of using

this macro is inlining the second stage operations at the place of usage. This way the `<[ ]>` brackets can be considered as markers of second stage computation.

The other way of understanding the example is by its implementation in Nemerle compiler. It is a function generating code, which will be substituted at the place of its usage. The `power1` function simply generates the code of arithmetic expression composed of multiplications and square operations. As mentioned earlier we utilize the fact, that `n` is independent from `x` and basing on that we know exactly how resulting expression should look like. Here we can view `<[ ]>` brackets as what they really are - syntax for constructing parts of new Nemerle code.

The macro above can be easily used to create specialized function

```
def power74 (x)
{
  power1 (x, 74)
}
```

or directly in code (which would yield direct in-lining of expressions and might be not a good idea for large chunks of generated code).

4.2 Imperative code generation

The example from previous section was easy to stage, because of its functional implementation style. The function body is composed of the conditions and recursive calls, but control flows always from top to bottom (no other jumps are performed). This is crucial for staging, because we can mark sub-trees in some branches of flow tree to be deferred to next stage as shown in figure 2.

Also, it is feasible only if we can divide the operations in the tree into two stages with a horizontal line. Those to be performed at first stage should be positioned in the upper half, so after executing them we can return expression describing the rest of operations from lower half. Those will be computed at the next stage.

Unfortunately sometimes the algorithm we work on isn't structured this way. Operations from different stages are interleaved, so we cannot do any computation before we need some of the dynamic input. Such a problem often arises when algorithm uses elements of imperative programming, that is computing expressions for side effects. For example imperative loop, whose side effect is to sequentially perform operations in the body, is entirely dependent on their stage. Expressions, which will be staged should always finish the control flow, but as loop can be repeated, they are not, thus the whole

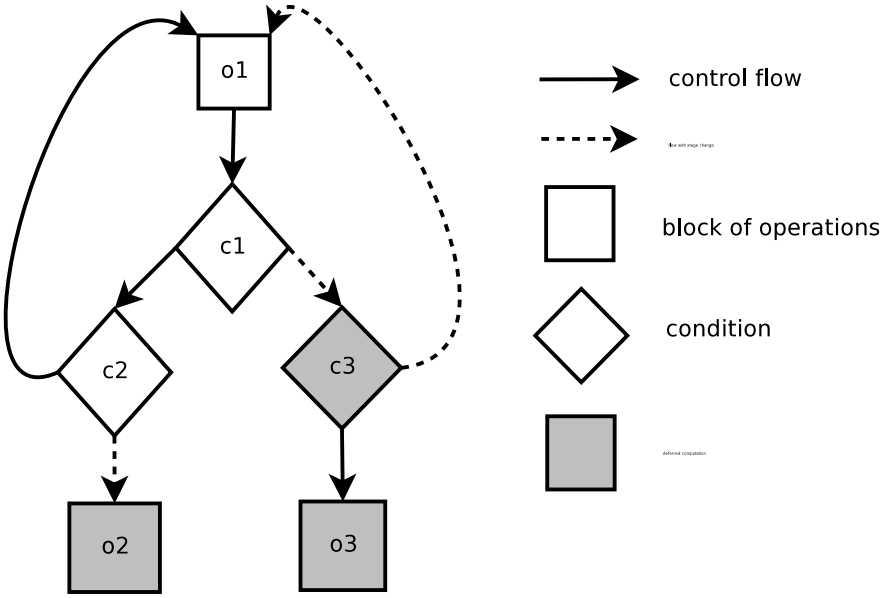

Figure 2: Control flow of the example function. Operations in white boxes are performed at the first stage (*o1, c1, c2*), while grayed (*c3, o2, o3*) in second stage. Dotted arrows denotes changing the stage.

loop must be deferred until next stage and we might loose some opportunities for partial evaluation.

The only solution not requiring a total rewrite of the algorithm is to explicitly store parts of generated code in some collection and at the end create final computation from it. For example the mentioned loop can be performed at first stage and expressions emerging from its unrolling are collected and then inlined into generated code.

The drawback is that staged version of program becomes more complex this way. Still, imperative approach often simplifies implementation of many algorithms and is easier to understand. We show that the negative impact of imperative style on staging (and partial evaluation) is not that great. Especially the ability to generate code imperatively allows us to handle it conveniently, which might be impossible at all in more conservative meta-programming systems.

4.2.1 Permutation function

Let us consider a function for computing permutation of given array. It takes input array as first parameter and array specifying the permutation as second parameter:

```
permute (data : array [int], permutation : array [int]) : void { ... }
```

where `permutation` array describes permutation as positions at which elements of input should appear in output (like for input $3, 2, 1$ and permutation $2, 1, 3$ the output will be $2, 3, 1$). The algorithm utilizes the fact that this representation directly exposes cycles of permutation and to permute the elements we must simply move each of them by one position on its cycle. It is presented below

```
permute (data : array [int], permutation : array [int]) : void
{
  def visited = array (permutation.Length);

  // we visit each cycle once using this top loop
  for (mutable i = 0; i < permutation.Length; i++)
  {
    mutable pos = i;
    // we walk through one cycle
    while (!visited [pos])
    {
      visited [pos] = true;
      // moving its elements by one position
      def next_pos = permutation [pos];
      unless (visited [next_pos]) {
        data [pos] <-> data [next_pos];
        pos = next_pos;
      }
    }
  }
}
```

As we can see this algorithm perform operations on `data` only in one line

```
data [pos] <-> data [next_pos]
```

which is the swap operation on elements of array. The rest of its steps is performed only basing on contents of `permutation`. This quickly leads us to the conclusion, that if we statically know this array, we can have highly optimized version of `permute`, which performs only sequence of swaps.

First we must obtain the value of permutation array inside our first stage function (a macro). The simplest way would be to store it in some static field visible to the macro, but we can also decompose the syntax tree of expression initializing an array, so it can be passed directly as macro's parameter. Second problem is that original permute function uses imperative style, so we must defer computation also in this style, explicitly building sequence of final computations.

```
macro permute1 (data, p_expr)
{
  def permutation = expr_to_array (p_expr); // new

  def visited = array (permutation.Length);
  mutable result = [];                      // new

  for (mutable i = 0; i < permutation.Length; i++)
  {
    mutable pos = i;
    while (!visited [pos])
    {
      visited [pos] = true;
      def next_pos = permutation [pos];
      unless (visited [next_pos]) {
        result = <[
          $data [$(pos : int)] <-> $data [$(next_pos : int)]
        ]> :: result;                       // new
         pos = next_pos;
      }
    }
  }
  <[ {..$result } ]>                        // new
}
```

As we can see the function didn't change much. We had to add the variable `result`, which is the list of resulting expressions to execute. It is used at the end in `<[ {.. $result } ]>` expression - the sequence of swaps on `data` is the result of macro (note that the expressions are in reversed order, but in this example it does not change the result of function).

`permute` and `permute1` can be used as follows:

```
Run (data) {
  def permute_specialized (data) {
    permute1 (data, array [10, 7, 11, 0, 12, 5, 6, 9, 4, 2, 1, 8, 3]);
  }

  def perm = array [10, 7, 11, 0, 12, 5, 6, 9, 4, 2, 1, 8, 3];
  permute (data, perm);
  permute_specialized (data);
}
```

4.3 Transforming interpreter into compiler

Partial evaluation introduces a novel approach to compilation and compiler generation. We can specialize an interpreter with a given program to obtain the compiled target program, which no longer require interpreting to run.

Such partial evaluator can be seen as a compiler of the language, for which the interpreter was written, into the target language of evaluator.

$$(Interpreter(p, input), p_1) \rightarrow p_1 compiled(input)$$

This process is often called the *first Futamura projection*, initially reported in [17]. Like mentioned before, partial evaluation implemented through meta-programming do not allow automating the process, so we cannot simply use interpreter and specialize it with given program. We can derive the new implementation by decorating interpreter code with staging annotations, explicitly specifying which operation should be performed at compile-time and which should be left to run-time. This approach gives much more freedom in changing the code which will be generated. With automatic partial evaluator we would depend completely on implementation of the interpreter (and partial evaluator) and had to accept that target program would always perform the subset of exact operations of the interpreter. This doesn't give much place for customizing or experimenting with code generators. We might also loose some opportunities for optimization as we show later.

In this section we present an example of creating specializable interpreter, by rewriting an interpreter of small language into a macro equivalent to the language's compiler. The model for operations performed in the language will be the following class representing a robot.

```
class Robot
{
  Orientation : int;
  X : int;
  Y : int;

  IsDown : bool
  {
    get { Orientation == 1 }
  }
}
```

It serves as an abstract state machine, which will be modified during execution of interpreted program. It could be arbitrarily extended without influencing described method.

Let us consider the set of terms

$$Expr ::= move(I) \mid left \mid right \mid value(S) \mid if(Expr, Expr, Expr)$$

which is represented by the algebraic type `Expr`

```
variant Expr {
  | MoveBy { steps : int; }
  | Left
  | Right
  | Value { prop : string; }
  | If { cond : Value; then : Expr; els : Expr; }
}
```

Program in this language is just a list of `Expr` terms. In real-life application they could be created from textual representation by some automatically generated parser targeting the `.NET` platform. In the rest of this section we focus on what happens after such an abstract syntax tree is generated.

4.3.1 Standard interpreter

Considered language is composed mostly of statements modifying the state of `Robot` object, which is our model of virtual machine (or environment). The only expression yielding a value is `Expr.Value` used inside the `Expr.If` condition, but it also depends on environment. Of course we could implement a simple lambda calculus here, like in [35], but we find our example simpler and still presenting the equivalent technique.

The interpreter is traversing structure of supplied expression and apply requested operations to the robot object. The interesting part is the `check_value` local function, computing value of property with given name (supplied as string) by querying the object at run-time. In the staged version we will be able to make this dynamic operation static and in consequence much faster.

The rest of interpreter implementation is straightforward and is roughly equivalent to the denotational semantics of the language.

```
Run (obj, expr)
{
  def check_value (val) {
    // we rely on .NET run-time reflection capabilities
    // to fetch value of property with given name
    System.Convert.ToBoolean (obj.GetType ().
      GetProperty (val.prop).GetValue (obj, null))
  }

  match (expr) {
    | Expr.MoveBy (steps) =>
      match (obj.Orientation) {
        | 0 => obj.X += steps
        | 1 => obj.Y += steps
        | 2 => obj.X -= steps
```

```
        | _ => obj.Y -= steps
      }

    | Expr.Left => obj.Orientation = (obj.Orientation + 3) % 4;
    | Expr.Right => obj.Orientation = (obj.Orientation + 1) % 4;

    | Expr.Value as val => _ = check_value (val)

    | Expr.If (val, e1, e2) =>
      if (check_value (val))
        Run (obj, e1)
      else
        Run (obj, e2)
  }
}
```

4.3.2 Staged interpreter

The problem with standard interpreter is that it must traverse expressions at run-time, which costs time. This is especially painful for programs, which do not change in time and are known at compile-time. The standard approach for removing this overhead is to write a compiler for the language. It would transform every input program to some target language (usually the native machine code). Unfortunately compilers have traditionally required much more skill and time to implement correctly than interpreters.

In this section we show how to obtain the same result by creating the *staged interpreter*. We divide computation performed by interpreter into two stages. First one is the preprocessing phase doing the traversal of input expressions. The other one consists only of actions specified in program, but do not involve any manipulation of expressions describing it. The first stage is independent of second one and can be completely executed before the run-time.

In case of our language the process is composed of annotating the standard interpreter with code quotations and splices (in multistage programming referred to as *staging annotations*). The recursive nature of our interpreter implementation allows us to think about the process purely as staging the program. It is not always the easiest approach though, as shown in section 4.2, but here this abstraction is very elegant and useful. The operations, which must be deferred to run-time execution are those dependent on run-time data, that is the environment (`Robot` object). We enclose them with `<[ ]>` in function below.

```
GenerateRun (obj, expr)
{
  def check_value (val) {
    <[ $obj.$(val.prop : dyn) ]>
  }

  match (expr) {
    | Expr.MoveBy (steps) =>
      <[ match ($obj.Orientation) {
           | 0 => $obj.X += $(steps : int)
           | 1 => $obj.Y += $(steps : int)
           | 2 => $obj.X -= $(steps : int)
           | _ => $obj.Y -= $(steps : int)
         }
      ]>

    | Expr.Left => <[ $obj.Orientation = ($obj.Orientation + 3) % 4 ]>;
    | Expr.Right => <[ $obj.Orientation = ($obj.Orientation + 1) % 4 ]>;

    | Expr.Value as val => <[ _ = $(check_value (val)) ]>

    | Expr.If (val, e1, e2) =>
      <[ if ($(check_value (val)))
           $(GenerateRun (obj, e1))
         else
           $(GenerateRun (obj, e2))
      ]>
  }
}
```

Note that `check_value` does no longer rely on dynamic invocation of property getter to obtain the value. It is now statically type-checked and compiled to more effective run-time constructs (no need for reflecting members of given object at run-time). It is a common case where more static information about program yields additional optimizations, not available in standard interpreter otherwise.

4.3.3 From interpreter to compiler

We have created two implementations of expression interpreter, but the second one operates on deferred computations (program fragments). Their usage differ, in our model only the macro can be consumer of program fragment.

In order to use single stage interpretation we run it consecutively on each expression from the list (a program). The staged interpreter is executed inside macro, which yields running first stage at compile time and generates program fragment containing the whole second stage computation.

Both functions use the external function `Scripts.GetScript` to load the program containing list of expressions to be interpreted against supplied `Robot` object.

```
Run (obj, name : string)
{
  def script = Scripts.GetScript (name);
  _ = script.Map (fun (e) { Scripts.Run (obj, e) });
}

macro GenerateRun (obj : PExpr, name : string)
{
  def script = Scripts.GetScript (name);
  def exprs = script.Map (fun (e) {
    Scripts.GenerateRun (obj, e)
  });
  <[ { ..$exprs } ]>
}
```

One of the most important features of our design is that at the place of use, calling macro looks like standard function call. This way, after defining both our interpreters we can use them with the same syntax.

```
def robot = Robot ();
Run (robot, "script1");
GenerateRun (robot, "script1");
```

The generated machine code and performance differs largely. The `GenerateRun` macro in-lines much more efficient code, which is semantically equivalent to running single stage interpreter.

5 Top level macros

The large novelty of Nemerle meta-system, especially in context of industrial usage is how it handle object-oriented layer of language. We have incorporated concepts of hygiene and code quotations to the domain of method and class definitions. It brings concept of program generation to the object hierarchy of program. This proves a very useful tool, where OO methodology implies usage of common design patterns and / or requires writing similar code in many places.

The example could be the need to automatically generate wrappers for some library, methods constituting some design pattern or entire object hierarchy from database schema.

Our experiments show, that the most useful macros for operating on class hierarchy requires much wider interoperability with compiler than simple syntax directed transformations on its data-structures. This is why we decided to expose the compiler API for usage inside macros. This approach allows user to query compiler about inheritance relations, types of members in given class, etc.

5.1 Language extended with OO components

We now describe the subset of object-oriented syntax of Nemerle language.

⟨*program*⟩ ::=
    | ⟨*class definition*⟩ ⟨*program*⟩

⟨*class definition*⟩ ::=
      `class` ⟨*var*⟩ [`:` ⟨*var*⟩ (`,` ⟨*var*⟩)*] `{` ⟨*class member*⟩ * `}`
    | `interface` ⟨*var*⟩ [`:` ⟨*var*⟩ (`,` ⟨*var*⟩)*] `{` ⟨*class member*⟩ * `}`

⟨*class member*⟩ ::=
      ⟨*var*⟩ `;`
    | ⟨*var*⟩ `(` [⟨*var*⟩ (`,` ⟨*var*⟩)*] `)` ⟨*block*⟩
    | `this` `(` [⟨*var*⟩ (`,` ⟨*var*⟩)*] `)` ⟨*block*⟩

⟨*expr*⟩ ::=
    | ⟨*expr*⟩ `.` ⟨*var*⟩ // member access
    | `this` `.` ⟨*var*⟩
    | ⟨*expr*⟩ `=` ⟨*expr*⟩ // assignment expression

For *program* and *expr* non-terminals only the new rules are specified.

The semantics of classes, interfaces and member access is the same as in C# or Java. Each class definition implicitly introduces its constructor functions (methods with a name *this*) to the scope. Those functions create values of given class' type, whose can be then used to access fields or invoke methods. *this* expression is used to access members of current class instance (it is also the same as in C#).

This object model allows for simple class sub-typing. In our simplified language we assume all members are publicly accessible and inherited in subclasses.

5.1.1 Stages of class hierarchy building

Analysing object-oriented hierarchy and class members is a separate pass of the compilation (as mentioned in section 3.1). First, it creates inheritance relation between classes, so we know exactly all base types of given type. After that, every member inside of them (methods, fields, etc.) is being analysed and added to the hierarchy. After that also the rules regarding implemented interface methods are checked.

For the needs of macros we have decided to distinguish three moments in this pass at which they can operate on elements of class hierarchy. Every macro can be annotated with a stage, at which it should be executed.

- *BeforeInheritance* stage is performed after parsing whole program and scanning declared types, but before building sub-typing relation between them. It gives macro a freedom to change inheritance hierarchy and operate on parse-tree of classes and members
- *BeforeTypedMembers* is when inheritance of types is already set. Macros can still operate on bare parse-trees, but utilize information about subtyping.
- *WithTypedMembers* stage is after headers of methods, fields are already analysed and in bound state. Macros can easily traverse entire class space and reflect type constructors of fields, method parameters, etc. Original parse-trees are no longer available and signatures of class members cannot be changed.

Later on we refer to these stages when describing class level macros.

5.1.2 Changes to meta-system

As we said in previous section, we want to algorithmically generate program fragments not only for expressions as before, but also for elements of class hierarchy. The syntax of `<[ ]>` quotations must be extended to distinguish those elements from core expressions (that is because ambiguity between function call and method definition syntax).

The additional rule for <*expr*> production is

⟨*expr*⟩ ::=
| `<[ decl :` (⟨*class declaration*⟩ | ⟨*class member*⟩) `]>`

Macro invocation also differs, because class-level syntax does not include *calling* defined elements. Classes and their members are just declarations.

Although, C# and Nemerle does include a clean syntax for *annotating* members with meta-data. We have incorporated this idea to specify that given element should be transformed by a macro.

The *class definition* and *class member* productions are extended to include this new syntax:

⟨*class definition*⟩ ::=
    | `[` ⟨*expr*⟩ `]` ⟨*class definition*⟩

⟨*class member*⟩ ::=
    | `[` ⟨*expr*⟩ `]` ⟨*class member*⟩

Where expression inside `[ ]` is expected to be the macro invocation.

The example of class declaration with macro annotations is

```
[ClassMacro ()]
class House : Building
{
  HasGarage;

  [FieldMacro ()]
  Inhabitants;

  LivingCosts (cost_per_person)
  {
    Inhabitants * const_per_person
  }
}
```

Expansion of class-level macros is performed at class hierarchy building phase of compilation (defined in section 3.1). It involves similar technique as expression macros expansion, but here every executed macro is supplied with an additional parameter describing an object, on which macro was placed. This way it can easily query for properties of that element and use compiler's API to reflect or change the context in which it was defined (as mentioned at the beginning of section 5).

For example a method macro declaration would be

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
                     Nemerle.MacroTargets.Method)]
macro MethodMacro (t : TypeBuilder, m : MethodBuilder, expr)
{
  // use 't' and 'm' to query or change class-level elements
  // of program
}
```

Macro is annotated with additional attributes specifying respectively the stage in which macro will be executed (one of defined in section 5.1.1), and the macro target.

5.2 Implementing design patterns

As a motivation for macros operating on object hierarchy we present an example how usage of design patterns can be automated and simplified.

Design patterns [18] are meant to ease programmers' work by distinguishing various common schemes of creating software and solving most often occurring problems. They identify and name patterns of interaction between objects in program and then suggests them as widely accepted solutions for problems.

The downside of many design patterns is that they often require implementing massive amount of code, which tends to be the same most of the time. Some patterns just point out how particular objects could communicate or what should be the inheritance hierarchy of classes representing given model. These patterns are just hints for programmers what is the preferred way of structuring their programs in chosen situations. Unfortunately others often imply large non-creative work for programmers trying to follow them. This is barely a waste of their time and leads to increased maintain costs.

With meta-programming approach, we could just write a generator for all methods, wrappers, auxiliary instances, etc. which need to be created in particular design pattern.

5.2.1 Proxy design pattern

PROXY pattern is based on forwarding calls to some object A into calls to another object B. Usually the object B is contained in an instance field of A.

The point of this is to imitate behavior of B in a new class. Usually we can just use derived class, but in case we do not have a direct way of creating B, but obtain existing instance externally (e.g. remote object on server, etc.), we have to query it in every operation. Proxy pattern allows us to promote instance of such object to the class, which now can also be extended with a new functionality. For example, we can implement B's interface in A (by simply passing all calls to B) and then override some of these methods with a new behaviour in derived class.

Suppose we have a class

```
class Math : IMath
{
  public Add(x : double, y : double ) : double { x + y; }
```

```
  public Sub(x : double, y : double ) : double { x - y; }
  ...
}
```

implementing `IMath` interface. Now we create a `MathProxy` class, which will provide full functionality of `IMath` by using externally created instance of `Math`.

The point is that forwarding every call in `IMath` requires a considerable amount of code to be written. We will use a macro, which generates needed methods automatically.

```
// Remote "Proxy Object"
class MathProxy : IMath
{
  // the stubs implementing IMath by calling
  // math.* are automatically generated
  [Proxy (IMath)]
  math; // object of type Math

  this()
  {
    math = ObtainExternalReferenceToMath ();
  }
}
```

From the user's perspective `MathProxy` can be used just like it was a subtype of `Math` but no additional methods must be written - in a standard approach programmer must manually write methods like:

```
Add( x, y ) {
  math.Add (x,y);
}
```

The implementation of `Proxy` macro is presented below:

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.WithTypedMembers,
                     Nemerle.MacroTargets.Field)]
macro Proxy (t : TypeBuilder, f : FieldBuilder, iface)
{
  // find out the real type specified as [iface] parameter
  def interfc = BindType (iface);
  foreach (meth in interfc.GetMethods ())
  {
    // prepare interface method invocation arguments
    def parms = meth.GetParameters ().Map (fun (p) {
      <[ $(p.ParsedName : name) ]>
    });
```

```
      // prepare function parameters of created method
      def fparms = parms.Map (Parsetree.Fun_parm);

      // define the wrapper method
      t.Define (<[ decl:
        $(meth.ParsedName : name) (..$fparms) {
          this.$(f.ParsedName : name).$(meth.ParsedName : name) (..$parms)
        }
      ]>)
    }
  }
```

Ignoring some of the implementation details (like creating program fragments for invocation arguments and method parameters), this macro is a simple function iterating over given interface's methods and adding wrappers for them to current class.

Moreover, macro can be easily customized and extended, so for example it can mark members with specified attributes, omit some of them, etc.

5.3 Hygiene considerations

Class level constructs of the language bring new problems regarding hygiene and how symbols are bound. When we consider member access operation, especially with ability to omit `this.` part for members of current class, we need to distinguish what the *current class* is and how it relates to hygiene of macro created symbols.

In this section we describe our design and implementation of class level hygiene. Some of the decisions might sound unintuitive and needs special explanation.

5.3.1 'this' reference

First of all, variables accessed through explicit usage of `this`, like `this.x` are always bound to the members of class, where generated code resides. This is not only the most intuitive semantics, but roughly the only valid one. One could imagine that the name should be looked up in its own context (its originating macro declaration, as described in section 3.4), but the only point of using `this` is to specify that name should be bound to the member of *class specified by* `this` reference. Now, we must take a look at what class `this` may point to. As macros are independent of class hierarchy (and in particular they are not instance methods), there is no valid class at the place of macro declaration, which created the discussed expression. The only possibility at

this point is to use lexically visible class at the place where generated code finally gets expanded and requiring that it is inside instance method.

5.3.2 Class symbols

For the plain symbols at expression level the standard scoping mechanism referred to in section 3.4 must be extended to consider references to class names and members. `x` might bind to the local variable (generated by a macro or locally defined) or to the member of current class (in which case it is the same as `this.x`). Local variables have precedence in this algorithm, that is if both local variable and class member with correct name are visible, then the former symbol is chosen.

Standard hygiene rules apply here - only definitions in the same context (macro color) as analysed symbol are considered visible. But in case of the class members, the search is performed only if the symbol's color matches the current class' color.

Note, that we do not check the context of candidate class member's declaration, but class itself. This is because members are always accessed through object instance, not directly. Thus, we just require that the considered symbol reference was generated together with the class it relates to. The reason behind such a design is visible in following example

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance,
                     Nemerle.MacroTargets.Class)]
macro AddMethod (ty : TypeBuilder) {
  // add a new method to the current type
  ty.Define (<[ decl:
    my_method (x) { x }
  ]>);
}

...

[AddMethod]
class A {
  foo () { my_method (1) }
}
```

If we required the context of `my_method` declaration and usage to be the same (which is not true in above example, because `my_method` is created inside the macro), it would lead to quite unintuitive behaviour. On the other hand, if we think about class members purely as meta-data associated with a class, then our approach seems reasonable. The important condition is that class' context must match the used symbol. This way the following code will work as expected

```
macro inject (body) {
  <[ def f (x) { x + 1 } f ($body) ]>
}
...

class B {
  f (y) { inject (y) }
}
```

Here `f` introduced by a macro is hygienic and the generated expression `f (y)` will be resolved to the local function call, not recursive call to method `B.f`. The algorithm described above ensures such a behaviour, because `B` has a different color than generated `f`.

5.3.3 Implementation

The implementation of the class level symbols coloring is an extension of the rewrite rules from section 3.4. We apply the existing rules (especially *Mon* and *MonVar* to the terms describing the added language elements: class definitions, class members and expressions).

The *Expand* rule can apply to the macro invocations present in attributes. The only modification here is that we pass additional parameters to the *Run* method.

The only nontrivial modification needed is the symbol lookup after all the macro expansion and coloring transformations are done. For symbol reference *Ref(id,col,g)* it is bound to the declaration from one of the following cases (in decreasing priority):

- If it is a part of member access expression, first the object part is looked up and it is expected to have a type of class containing *id* member.
- There exists a local value (variable or function) declaration in scope of current class method or global program with name *id* and color *col*.
- The current class' color is equal to *col*, it contains *id* member and the current class method is an instance method.
- There is a static method or class visible in g global environment.

If none of the above cases are valid, an error is issued.

5.4 Aspect-Oriented programming

Aspect-oriented programming was first proposed [25] as a solution for code "tangling", where the distinct *aspects* of design need to be merged manually in real programs to achieve efficient implementation. The problem arises when there are several *cross-cutting concerns* in one fragment of program (e.g. like synchronization, logging, iterating over data and modifying data), which would otherwise be written separately. In large programs there is always a trade-off between code simplicity (no tangling) and efficiency (manually optimized code).

The idea of *AOP* is to program operations concerning each of the single aspects of design in a single place and then have an automatic *aspects weaver* to combine them into the working program. The initially presented design [25] of this operation resembles a kind of macro transformation. There is so called *component language* in which the base structure and functionality of program is coded (like the used functions, classes, etc.) and *aspect languages*, which specify how some additional code is merged into the base program. *Aspect weaver* plays a role of macro expander.

In further research, operations available in aspects were constrained and finally they lost the character of general programs operating on code fragments. They became a sort of specification language [11] for modifying code and classes in a predesigned way, like in AspectJ [24]. Currently the concepts introduced in AspectJ are the standard vocabulary in *AOP* literature and include following elements of aspect language:

- *pointcut* - Aspect-oriented languages allow to specify a set of execution points in program (*join points*), where cross-cutting behaviour is required. These points can be altered by the aspect-weaver to introduce behaviour specified in aspects.

- *advice* - A piece of advice is code that executes at a pointcut. Advices are inserted into pointcuts according to the specification introduced by aspect.

- *aspect* - Aspect is a set of specifications of how to insert advices into the program code or modify it with introductions.

- *introduction* - An introduction is a programming element, such as an attribute, constructor or method, which can be added by aspect to the given type extending or modifying it. Also the object-oriented hierarchy of classes can be modified.

Nemerle meta-programming system together with exposed compiler's API can be used to implement aspect-oriented model. This would lead to the compile-time aspect language (as opposed to byte-code manipulation utilized in AspectJ [21]) and easier visualisation of programs created during consecutive weaving of aspects.

On the other hand, meta-programming allows much more than a predefined insertions of code and it could be used to generalize the *AOP* approach. Our experience suggests that algorithmic code transformation is a convenient and simple way of solving similar problems. Also using custom attributes to mark methods and classes for transformation is a more localised (and thus easier to read and understand) way of specifying the pointcuts.

6 Syntax extensions

Usually programming languages have predefined fixed syntax, which is core part of their specification. Some of them (like Lisp [31] and Scheme [1]) use a very uniform and flexible notation, which allows specifying new syntax constructs in a convenient way. Others (like Camlp4 [13], OpenJava [37], Java Syntactic Extender [4]) allow extending some fixed grammar entries.

Cardelli studies [10] extensible grammars in a more formal way, where user is allowed to define new non-terminals and extend existing ones. The new grammar is then statically type-checked for unambiguity and consistence. The LL(1) parsing functions are then generated.

The motivation behind the syntax extensions is to give user ability to extend the language and customize it with the needs of particular project. There is a raising need for embedding domain-specific languages into the general purpose ones [8]. They are especially useful in presence of general macro system, where user can specify semantics of new syntax and embedded language in a convenient way.

Nemerle has built-in syntax extension capabilities. In our design we decided to rely on a rather ad hoc approach, which gives less static assurances, but is easier to use and at the same time more flexible in most of the use cases.

Extensions in Nemerle are limited to the fixed places of language grammar (like in Camlp4), but they have the ability to defer parsing of entire fragments of token stream. This gives convenient tool for adding simple syntax constructs and at the same time a little more effort (supplying an own parsing function) allows embedding entirely new language into Nemerle.

6.1 Base system

Base system of syntax extensions is quite simple and easy to use, but also lacks expressibility. Every macro can be marked with one or more syntax patterns. *Syntax pattern* is a list of tokens and grammar elements expected to appear in parsed input token stream to constitute given extension. It defines a new production for one of the non-terminals in a language grammar.

After such a macro is loaded, compiler adds syntax patterns defined for it to a special tree describing extended language syntax. The tree is build dynamically during parsing, based on the set of imported namespaces. New productions are merged into the tree - common prefix of productions is represented by a single path, while the first token on which they differ causes branch. If there is a node, where two productions end or the branch occurs on two non-terminals, which do not have distinguishable first valid characters, parser cannot decide which path should be chosen. Error is though reported only if user tries to parse the code, which is ambiguous. This is our design decision, syntax extensions are dynamic, although we could easily analyze the grammar when merging new syntax, so opening two namespaces with conflicting syntax would yield an error. This would make syntax extensions more limited, for example one opens two conflicting namespaces, but only for the purpose of using disjoint syntax patterns. In this case we would forbid completely valid code.

As an example of described technique consider following two macro definitions:

```
macro if_then (cond, expr)
syntax (``if'', ``('', cond, ``)'', expr)
{ ... }

macro if_then (cond, expr1, expr2)
syntax (``if'', ``('', cond, ``)'', expr1, ``else'', expr2)
{ ... }
```

Syntax patterns have common prefix, so they are merged into tree-like pattern presented in figure 3.

Parser always chooses the longest matching path when processing input. Thus, for the *if/else* example the *else* option will be used when possible.

The algorithm described above is just a simple transformation performed in most parser generators [2] when analysing productions for single non-terminal. Its purpose in Nemerle is to be the base for more powerful deferred parsing described below and have the advantage that it do not confuse users with complex grammar based system.

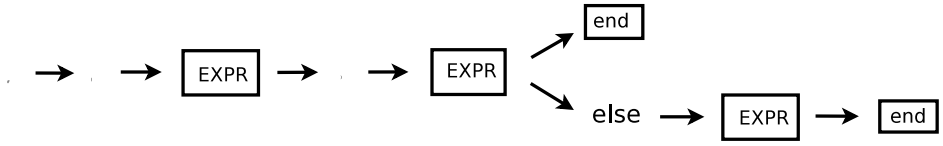

Figure 3: if/else grammar tree

6.2 Parsing stages

Base extensions system can be implemented easily in a standard recursive-descent parser. We must add a function, which walks the extensions tree while processing the input token stream. It should gather syntax elements defined in pattern (which will become actual arguments of the macro invocation) and when it get into the node which ends syntax pattern of macro m, then the macro invocation expression $m(p_1, \ldots, p_n)$ is returned.

But in order to embed arbitrary language inside Nemerle grammar we need the more general way of extending parser. In particular we would like to specify that given fragment of program should be parsed by a completely different function. User should be able to easily specify that function and the part of code to be parsed with it.

To solve the problem we have created an additional preparsing phase (mentioned in section 3.1), which processes token stream from lexer before it is analysed by actual parser. Its main task is to match into groups the commonly used open and closing brackets (i.e. `()` parentheses, `{}` braces, `[]` and `<[ ]>` brackets). Every bracket pair composes a token group.

Definition 7 (Token group) *The node of the token tree created by preparsing stage, containing representation of token stream fragment. It contains a list of tokens and token groups. Every bracket pair is divided into token groups separated by* separator *token specific to given brackets kind (period* `,` *for* `()` *and* `[]`*, semicolon* `;` *for* `{}` *and* `<[ ]>`*).*

This preprocessing stage helps also in reporting parse errors. Mismatched parentheses are the very common mistake in program and we can deal with it by reporting which exact brackets do not match. Also error recovery is later easier, in case of error we can just skip processing of entire token group and continue.

Summarizing, there are three stages before we get the parse-tree representation of textual program. First standard scanning phase transforms program

text into a stream of tokens, after that pre-parsing phase creates the tree of token groups and finally the main parser builds the program fragment, which is then used in macros and typed.

6.3 Deferring parsing

The most important advantage of dividing input in described way is that we can separate the whole token group from parsing by main parser and use completely different algorithm to analyse it. This solves one of the posed problems, that is specifying which fragment of program should be treated differently.

In order to process such token gropus insie macro, one of its parameters must be annotated with `Token` type. Such an argument can only be the result of parsing syntax extension, because standard program fragment can never be in unparsed form. Thus, if we do this, macro must also supply a syntax pattern containing given parameter name. When parser is processing the syntax pattern and if the next node is `Token`, then it takes the nearest token group from the input and places it as an argument for macro invocation node.

For example followin macro definition

```
macro xml_literal (tokens : Token)
syntax ("xml", tokens)
{
  // process 'tokens'
}
```

introduces syntax for parsing code like

```
...
def x = xml <person><name>John</name></person>;
...
```

The second of our problems (giving user the ability to parse distinguished part of program by supplied function) is immediately solved by allowing macro to be parametrised by token group. Inside macro, arbitrary functions can be used and token group can be parsed by routine specified there. We give user a full freedom here, but it is consistent with our design. There is already the assumption that macros are allowed to execute arbitrary code, here we only change the state of objects we pass to it. Deferred parsing is all performed by macros during their expansion.

The largest disadvantage of described method is that input stream is initially transformed before macro can analyse it and separator tokens have

special meaning here. For example we cannot use ; inside the xml literal presented above, because it closes the current token group. This leads to the following restriction on languages embedded using this technique: only languages conforming lexical and token tree structure of Nemerle can be parsed without modifying compiler itself.

6.4 Extending top level

The syntax extensions we have described above were all attached to the expression level macros. When we consider the language with object-oriented constructs described in section 5.1 it is reasonable to extend syntax also for them.

Note that macros operating on those constructs were till now specified inside custom attributes, which we have used to mark entities occurring at top level. We chose to maintain this design and make syntax extensions closely bundled to types and their members' declarations. The new syntax can occur at the fixed places in existing nonterminals' productions and are translated as annotating members with attributes. Alternatively we could make syntax extensions introduce their own elements to the abstract tree, which would be later expanded like macros. The only situation where this might lead to more expressibility is for programs or classes without any standard member. If there were any, attaching the macro to it and later ignoring additional parameter would result in the same effect.

We have also reused the system of syntax patterns specifying the list of grammar elements, which must occur in the input. In this case parsing of an extension is triggered at the several chosen places in grammar production constituting given entity. The most commonly used ones are at the beginning of production and between the header and body of standard class member. For example it is possible to use new syntax before methods and types or after their declaration's prologue.

These hard-coded rules are meant to imitate idea of decorating members with new meta-data, just like custom attributes. Such approach can also be seen in OpenJava [37].

Consider two examples of top level syntax extensions.

```
[Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance,
                     Nemerle.MacroTargets.Method)]
macro Requires (_ : TypeBuilder, m : ParsedMethod, assertion)
syntax ("requires", assertion)
{ ... }

[Nemerle.MacroUsage (Nemerle.MacroPhase.BeforeInheritance,
```

```
                    Nemerle.MacroTargets.Method)]
macro Async (_ : TypeBuilder, m : ParsedMethod)
syntax ("async")
{ ... }
```

They modify the body of method to include respectively the validation of given boolean assertion and making it executed asynchronously. They can be used in a following way:

```
class Foo {
  Bar (x) requires (x > 0)
  { ... }

  async Execute (name)
  { ... }
}
```

7 Type reflection

As we have mentioned in the introduction, macros in statically typed language should offer more than simple syntactic transformations. They should be able to reflect the static types of supplied expressions in order to generate highly specialized code. In languages like Lisp there are no types at all and macros are always limited to reflect only syntactic structure of input. In some interesting applications this is not sufficient and would not allow macros to be used for tasks, which are perfectly suited for them. The problems arise especially when we must deal with assumptions and design of target platform similar to `.NET`, for example division of all objects into value-types and reference types, where comparison to `null` is valid only for reference types and macro should be able to take it into account during code generation.

In Nemerle we have directly exposed some of compiler's API to the macros. This way we can analyse class hierarchy like in section 5, but also ask about the type of given expression. This is the natural approach and it works well in language with explicit type annotations at every declaration or with simple bottom-up type inference. Unfortunately in presence of more advanced inference, reflecting type of expression isn't independent from the rest of typing process. In order to enable this feature to work in general way we must interweave macro expansion with typing and allow macros' parameters to be marked as already typed code. This way compiler can handle macro expecting typed code in a special way.

Maya language This feature exists in Maya language [5] in a limited form. Maya allows annotating macro parameters with a structure and expected static type of input AST nodes. They can also be specified as unparsed bracket trees (in a similar way to our token groups). Then before particular expression is chosen to be expanded by some macro the analysis of its sub-expressions from left to right is being done. The delayed parsing is also performed. Finally when only one macro is applicable, it is expanded. Maya is a language without type inference and such approach is simpler to implement.

7.1 Type inference algorithm

To explain details of implementation we must first introduce the general description type inference engine used in Nemerle[27].

It is based on on-line constraint solving combined with deferral of certain typing actions. Compiler gathers sub-typing information about type variables in calls and builds the relation graph describing it. The graph is required to be consistent all the time, so any contradictious typing constraints are detected early and can be reported with proper location in source code. If typing algorithm finishes, there are two possible outcomes. When information stored in graph yields unambiguous type annotations for the program, then is succeeds. Otherwise an error occurs meaning that some expressions are ambiguous or there is not enough information for resolving their types.

In order to provide constraint solver with data, compiler processes declarations and expressions in methods' bodies. Each sub-expression receives a fresh type variable, which is added to the constraints graph. Algorithm proceeds with bottom-up analysis and updates current requirements on existing type variables. Obtaining type of some expression, for example member access or overloaded function call requires that types of their sub-expressions are known. If they are not, the typing of such node is deferred and stored into the queue. Compilation is then continued and more information is gathered.

The mentioned queue is called *delayed typings* queue and after the main typing pass described above finishes it is processed from the beginning. Each expression, which we are able to type here might give additional constraints on other expressions, and so on. The algorithm is thus executed until a fixed point is reached. If there are still some delayed expressions at this stage the error is reported.

7.2 Delayed macro expansion

The algorithm presented in previous section is perfectly suited for the needs of delaying macro expansion.

If one of the parameters of given macro is marked with `:TExpr`, then it is not expanded at the first step of main typing pass analysing the expression. It is added to the *delayed typings* queue instead and only arguments for which user requested type information are processed further as part of the type inference algorithm. This way macro gets expanded just after compiler gathers enough information for setting the final type information of specified parameters.

Consider the *foreach* macro mentioned earlier.

```
macro foreach (var, collection : TExpr)
{
  // collection contains full information about its type
}
```

Now its expansion is deferred until expression supplied as second argument is fully typed. It is then passed to macro as object of type `TExpr`, which is compiler's representation for nodes of correctly type-checked abstract program tree.

Note that the invocation of *foreach* macro do not bring any new information for the use of type inference engine. In that sense it is completely transparent for the engine and is treated as one more expression, which needs full type information to be compiled. Fortunately this information can come from other place, which is possible thanks to the constraints solver. For example

```
def f (col) {
  // col is of unknown type at this point
  foreach (x, col)
    print (x);
  1 :: col // but here we know col must be list[int]
}
f ([1, 2, 3]) // also this expression yields the type of col
```

7.2.1 Specifying type information

Sometimes we want to help compiler to infer the types. The standard way to do this in Nemerle is by type enforcement expression `expr : type`. Such an expression can also be generated by macro. But with macros requesting typed parameters we would like to specify their types as part of the macro declaration.

The syntax to do this is

```
macro if_macro (cond : TExpr [bool], e1, e2)
{
  <[ match ($(cond : typed)) {
       | true => $e1
       | false => $e2
     }
  ]>
}
```

We parametrize the `TExpr` with the expected type. This way compiler can immediately add a new tight constraint to the solver.

One more thing to note in above example it the use of `$(cond : typed)`. It is the additional splice syntax used for embedding already typed expressions (like those obtained in parameters with `:TExpr`).

7.3 Matching over type of expression

All the nodes of `TExpr` tree have type information associated with them. We can access it through the `.Type` property, which returns type variable (mentioned in section 7.1) bounded with given expression. In case of macro parameters the type variables are already fixed and concertised types represented by `MType` variant.

User can use standard pattern matching to decompose those objects. Macro can then generate code according to the structure of type. Also the utilities from compiler API can be used to check presence of given types in sub-typing relation, etc. The down side of this approach is that customizing even simple macro with type reflection needs considerable code to be written and moreover much knowledge about compiler's internal structures and API. Macro system should support choosing implementation according to the parameters' type more natively.

7.3.1 Macros overloading

To solve the problem we can extend the treatment of delayed macro expansions in a similar way to overloaded function calls. This way several macros can be defined varying only on the expected type of parameters marked to be included into typing. Such extension naturally fits into algorithm of choosing overload possibilities, because both of them require full type information before finishing.

In connection with delayed parsing this feature allows syntax of the language to be dependent on the static type of expressions involved. We can mark some macro parameters as `:TExpr` and others with `:Token`, the only

limitation here is that expressions specified for typing must occur as standard expressions in syntax pattern. This enforces some specific shape on patterns - token groups must be separated from expressions.

7.4 Attaching macros to class members

Another way of very selective macro expansion is a direct overriding of given class member reference. We want to specify that every use of method overload `foo(x : int` y : string), should be intercepted and instead of emitting the run-time function call some chosen macro should be inserted and expanded with original parameters of call supplied to it.

The idea (present also in OpenJava [37] or a special case of Maya [5] type-driven macros) is rather a simple extension of compiler, but in connection with code quotation and other features of Nemerle meta-system it presents considerable increase of expressibility.

As a motivating example consider a problem of *FLYWEIGHT* [18] design pattern mentioned in OpenJava paper [37]. It focuses on objects-sharing and requires that user should call a method of special *factory* module to obtain instances of some class instead of calling constructor directly. We would like the expression `Glyph('c')` to be replaced automatically by `GlyphFactory.CreateCharacter('c')`, but without introducing a `Glyph` macro, which would capture every call to `Glyph` (even the incorrect overloads), but would not work with qualified calls, like `MyApp.Glyph`. To do this, we write a macro

```
macro glyph_capture (parm)
{
  <[ GlyphFactory.CreateCharacter($parm) ]>
}
```

and attach it directly to the proper constructor of `Glyph` class.

```
class Glyph {
  ...
  [Nemerle.Macros.Attach (glyph_capture)]
  public this (c : char) { ... }
}
```

This feature is especially useful when we want an accurate and convenient tool for replacing some functions calls with alternative code (like in aspects-oriented programing).

8 Related work

In following sections we summarize related work on the topic of meta-programming, especially with features similar to Nemerle.

8.1 MetaML

MetaML [36] is a statically typed functional programming language with special support for program generation. It provides three staging annotations, which allow the construction, combination and execution of object-programs. The notion of type-safety of MetaML is strong: Not only is a meta-program guaranteed to be free of run-time errors, but so are all of its object-programs. It was developed mainly with multistage programming on mind, especially with focus on run-time code generation and cross-stage persistence.

The MetaML's type system unlike Nemerle's performs static typing of code under brackets. It introduces the notion of level of expression, which is modified by staging annotations. Values can be implicitly marked by compiler as cross-stage persistent, which is inferred by comparing the level at which variable is declared and used. MetaML is implemented purely in interpretive environment, which greatly simplifies handling of cross-stage persistence.

8.1.1 Static-typing vs run-time

The largest advantage of MetaML over Nemerle is the type-safety guarantees for generated code. Unfortunately this feature comes with several important trade-offs.

First, it limits the flexibility of code generation (as shown in section 4.2) by forbidding construction of code referencing variables not statically in scope of the quoted code. The property referred to as *closed code* forces user to use the language purely as multi-stage computation framework and not for program generation. Symbols used in object-program cannot be parametrized by meta-function input. This is particularly important when we want to apply meta-programming to object-oriented elements of program, like classes and their members, whose definitions are separated from places of use.

Another related issue is inability to break hygiene, which is very useful in implementing domain-specific languages. Again, in statically typed object-programs we expect every symbol to be bound at the compile-time of meta-program. But when some macro is able to break hygiene it is also able to introduce definitions, which can capture variable references in the context of

macro use, so they must be bound after macro expansion. This effectively prevents static-typing.

The approach we have chosen in Nemerle is not strictly throwing away idea of type-safety. Our system is a compile-time meta-programming framework, where all the program generation and transformation is being done before the actual object-code is executed. In a result, we type-check the transformed object-programs and it still happens before the run-time. We mostly lose the earlier error detection, which increases the development and debugging time. The real problem would occur only after using the system for run-time code generation. From the model point of view the terms *run-time* and *compile-time* we use here are just two stages of computation, to which we should also add the phase of macro compilation. But in real-life software development there is an important difference between the last stage (*run-time*) and all the previous ones - it is executed by user, while the others are run by developer of application. Thus the highest priority is to avoid as much errors as possible at the last stage. Our system fulfills this goal quite well.

Abandoning the strong typing approach seems to be the common way of increasing expressibility in compile-time meta-programming languages like Template Haskell [30].

8.2 Template Haskell

Template Haskell [30] does the interesting mix of ideas from MetaML with a more ad hoc approach - using algebraic data-types side by side with quotations for constructing code, delayed typing of object-code, breaking hygiene.

The brief list of differences and similarities to Nemerle macros is:

- Template Haskell resolves bindings during translating of quotations, which brings ability to reason about type-correctness of object code, before it is fully expanded. It allows detecting errors much earlier. Nevertheless, the presence of `$` splicing construct makes typing postponed to next stage of compilation, in which case new bindings must be dynamic.

- Template Haskell macros are completely higher-order, like any other function: they can be passed as arguments, partially applied, etc. This however requires manually annotating which code should be executed at compile-time. We decided to make macros callable simply by name (like in Scheme), so their usage looks like calling an ordinary function. We are still able to use higher-order functions in meta-language

(functions operating on object code can be arbitrary), so only the use of top meta-function (prefixed with `macro`) is triggering compile-time computations.

- Breaking hygiene is limited in Template Haskell mostly to variable references. Declaration splicing is allowed, but can occur only in top-level code and requires special syntax. We do not impose such restrictions, which seems a good way of taking advantage of binding names after macro expansion and imperative style present in Nemerle. It is natural for an imperative programmer to think about introduced definitions as about side effects of calling macros, even if these calls reside within quoted code.

- Some simple operations are possible to be performed on type declarations. We introduce general macros operating on object hierarchy, which are able to imperatively modify it. Moreover, they look like attributes attached to type definitions, so again programmer does not have to know anything about meta-programming to use them.

There are still many similarities to Template Haskell. We derive the idea of quasi-quotation and splicing directly from it. Also the idea of executing functions during compilation and later type-checking their results is inspired by Template Haskell.

An important observation is that usage of algebraic types for code construction and decomposition seems to be required in Template Haskell only for the purpose of solving a few specific problems with quotation syntax and for doing pattern matching on code. For example program constructs holding variable size data (like tuples, list literals, sequences, etc.) cannot be expressed in TH quotation syntax. Instead of using much more verbose data-type notation, we have introduced the `..$exprs` (where `exprs` holds list of expressions) syntax, which is similar to Lisp comma-atsign (`,@`) and creates an expression representing sequence of other expressions. Also extending pattern matching with quotation syntax is better than relaying on the direct matching on the nodes of internal compiler's data-structures.

8.3 Lisp

Lisp [31] had meta-programming capabilities in the early times when Timothy Hart introduced macros to the language [32]. It gave a powerful tool to the programmers, but often suffered from the problem of inadvertent name capture, which could be solved only by explicit usage of *gensym* in every

macro utilizing temporary variables. The problem was later addressed by Scheme hygienic macros.

The simple and uniform structure of syntax allows treating code as data in a convenient way. Nevertheless, Lisp uses a special syntax for constructing code. Its Nemerle equivalents are presented in following figure:

Name	Lisp	Nemerle
quasi-quote	`	<[ ]>
unquote	,	$
splice	,@	..$

8.4 Scheme

Scheme [1] is a statically scoped dialect of the Lisp programming language. It is the first programming language to support hygienic macros, which permit the syntax of a block-structured language to be extended reliably.

Our system has much in common with modern Scheme macro expanders [14]:

- Alpha-renaming and binding of variables is done after macro expansion, using the context stored in the macro in use site.
- Macros can introduce new binding constructs in a controlled way, without danger to capture third party names.
- Call site of macros has no syntactic baggage, the only place where special syntax appears is the macro definition – this implies simple usage of macros by programmers not aware of meta-programming.

Still maintaining the above features we embedded the macro system into a statically typed language. The generated code is type-checked after expansion. We also provide a clear separation of stages – meta-function must be compiled and stored in a library before use.

Works on Scheme last quite long, and many interesting features have been proposed. For example first-class macros in [7] seem to be possible to implement in Nemerle by simply passing functions operating on object code.

8.5 MetaOCaml

MetaOCaml [9, 20] is designed as an implementation of the basic multi-stage programming ideas from MetaML. It incorporates MetaML's annotation language and high-level code representation of delayed computations into OCaml language. It is a very interesting work on bringing multi-stage programming into a wider use.

The *Run* construct together and side-effects caused problems in static typing of meta-language and in initial paper [9] authors decided to always type-check generated code before running it. It was recently addressed in [33] by special analysis in type-system, which parts of program are runnable.

The implementation status of MetaOCaml shows the difficulty of performing run-time code generation using the native code compiler. It requires developing a framework similar to the `.NET` run-times, which would be able to generate, JIT compile and run code on fly.

8.6 MacroML

MacroML [19], the proposal of compile-time macros on top of an ML language. It is implemented as a translation to MetaML constructs and inherits its ideas of static typing the object-code. MacroML additionally enables pattern for introducing hygienic definitions capturing macro use-site symbols. It do not allow generating symbols, which could bind to declarations in context of macro use (like our `$(x : usesite)`), but allow reusing name supplied as macro parameter in its body. This isn't an answer for implementing domain-specific languages, but allow introducing new binding constructs. It is done without need to break typing of quotation before expansion.

Macros in MacroML are limited to constructing new code from given parts, so matching and decomposing of code is not possible.

8.7 CamlP4

CamlP4 [13] is a preprocessor for OCaml. Its LL(1) parser is completely dynamic, thus allowing quite complex grammar extensions to be expressed. Macros (called syntax extensions) need to be compiled and loaded into the parser prior to be used. Once run, they can construct new expressions (using quotation system), but they work only as parse tree level transformations. It is not possible to interact with compiler in more depth or perform general computations inside the transformers.

8.8 C++ templates

C++ templates [38] are probably the most often used meta-programming system in existence. They offer Turing complete, compile-time macro system. However, it is argued if the Turing-completeness was intentional, and expressing more complex programs entirely in the type system not designed for this purpose is often quite cumbersome. Yet, the wide adoption of this feature shows need for meta-programming systems in the industry.

There are a few lessons we have learned from C++ example. First is to keep the system simple. Second is to require macro precompilation, not to end up with various problems with precompiled headers (a C++ compiler must-have feature, because of performance).

9 Conclusions

We have presented an advanced compile-time meta-programming system incorporated into modern functional and object-oriented language. The system allows for convenient extending of core language with new constructs and syntax at both expression and class levels. It presents an unique approach of giving macros ability to reflect static information gathered by compiler. They have the property of automatic hygiene and are easy to use. The system proved its usefulness in implementation of Nemerle compiler and various macros proposed by users.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug. 1998.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[3] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[4] J. Bachrach and K. Playford. The Java Syntactic Extender (JSE). *ACM SIGPLAN Notices*, 36(11):31–42, Nov. 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[5] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. *ACM SIGPLAN Notices*, 37(5):270–281, May 2002.

[6] A. Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[7] A. Bawden. First-class macros have types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 133–141, N.Y., Jan. 19–21 2000. ACM Press.

[8] G. Bierman, E. Meijer, and W. Schulte. Unifying tables, objects and documents. To appear in proceedings of DP-COOL 2003, 2003.

[9] C. Calcagno, D. Genova, I. Roquencourt, L. Huang, Q. M. London, W. Taha, and X. Leroy. A bytecode-compiled, type-safe, multi-stage language, Nov. 21 2001.

[10] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital Equipment Corporation, Systems Research Center, Feb. 1994.

[11] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proc. 23rd Int'l Conf. Software Engineering (ICSE)*, pages 5–14, May 2001.

[12] B. E. Coffin. *The C++ Programming Language*, by Bjarne Stroustrup. *C Users Journal*, 6(6):63–??, June 1988.

[13] D. de Rauglaudre. Camlp4, a preprocessor for object caml, 2003. http://caml.inria.fr/pub/docs/manual-camlp4/.

[14] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, Dec. 1992.

[15] M. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. Technical Report TR-97-04-06, 1997.

[16] M. Flatt. Composable and compilable macros: you want it when? In *ACM International Conference on Functional Programming*, 2002.

[17] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, Dec. 1999.

[18] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[19] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. *ACM SIGPLAN Notices*, 36(10):74–85, Oct. 2001.

[20] M. Guerrero, E. Pizzi, R. Rosenbaum, K. Swadi, and W. Taha. Implementing DSLs in metaOCaml. *ACM SIGPLAN Notices*, 39(10):41–42, Oct. 2004.

[21] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 26–35. ACM Press, Mar. 2004.

[22] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2):3–27, 2000.

[23] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10):59–65, 2001.

[25] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[26] N. Linger and T. Sheard. Binding-time analysis for metaml via type inference and constraint solving. In *TACAS*, pages 266–279, 2004.

[27] M. Moskal. Type inference with deferral. Master's thesis, University of Wrocaw, Poland, 2005.

[28] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. The C programming language. Comp. Sci. Tech. Rep. No. 31, Bell Laboratories, Murray Hill, New Jersey, Oct. 1975. 31 Superseded by B. W. Kernighan and D. M. Ritchie, *The C Programming Language* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[29] T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer Verlag, 2001.

[30] T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. In M. M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, Oct. 2002.

[31] G. L. Steele. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1984. With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.

[32] G. L. Steele, Jr. and R. P. Gabriel. The evolution of Lisp. *ACM SIGPLAN Notices*, 28(3):231–270, Mar. 1993.

[33] Taha and Nielsen. Environment classifiers. In *POPL: 30th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2003.

[34] W. Taha. Multi-stage programming: Its theory and applications. Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, Nov. 1999.

[35] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

[36] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[37] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: A class-based macro system for java. In *OORaSE*, pages 117–133, 1999.

[38] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–38, 40, 42–43, May 1995.