Supporting Compile-Time Debugging and Precise Error Reporting in Meta-programs

Yannis Lilis[1] and Anthony Savidis[1, 2]

[1] Institute of Computer Science, FORTH
[2] Department of Computer Science, University of Crete
{lilis,as}@ics.forth.gr

Abstract. Compile-time meta-programming is an advanced language feature enabling to mix programs with definitions that are executed at compile-time and may generate source code to be put in their place. Such definitions are called meta-programs and their actual evaluation constitutes a compilation stage. As meta-programs are also programs, programmers should be supported in handling compile-time and runtime errors, something introducing challenges to the entire tool chain along two lines. Firstly, the source point of a compile error may well be the outcome of a series of compilation stages, thus never appearing within the original program. Effectively, the latter requires a compiler to track down the error chain across all involved stages so as to provide a meaningful, descriptive and precise error report. Secondly, every compilation stage is instantiated by the execution of the respective staged program. Thus, typical full-fledged source-level debugging for any particular stage should be facilitated during the compilation process. Existing implementations suffer in both terms, overall providing poor error messages, while lacking the required support to debug meta-programs of any staging depth. In this paper we firstly outline an implementation of a meta-programming system offering all mentioned facilities. Then, we detail the required amendments to the compilation process. Finally, we discuss the necessary interoperation points between the compiler and the tool-chain (IDE).

Keywords: Meta-programs, compile-time meta-programming, staged languages, source-level debugging, error messages.

1 Introduction

The term meta-programming is generally used to denote programs that generate other programs and was originally related to the existence of a macro system like the *C Preprocessor* (CPP) [1] or the *Lisp* macro system [2] that would allow program fragments to be built up at compile-time. Lexical systems like the CPP are recognized as being inadequate for meta-programming as they operate on raw text, unaware of any context information, while most languages do not share Lisp's syntactic minimalism to provide an equally powerful facility with seamless integration. In modern languages, meta-programming is closely coupled with functions that operate on some abstract syntactic form, like an abstract syntax tree (AST), and can be invoked during compile-time to change existing code or produce and inject additional code in the source being compiled. Such functions are called meta-functions and they as a whole constitute the meta-program. The compilation of a program that contains a meta-program requires it to be executed at compile-time to produce a possibly changed source file. If the resulting source contains additional meta-programs they are executed in the same way until we reach a final source with no meta-programs that will be compiled into the final executable. This iterative process may involve multiple steps of meta-program evaluations called *compilation stages*. Languages that support such a compilation scheme are called *multi-stage languages* ([3], [4]) with *MetaML* [5] and *MetaOCaml* [6] being two typical examples. Multi-stage programs are essentially programs whose source code is finalized through a sequence of evaluations defined in the program itself. They use special annotations to explicitly specify the order of their computations, with respect to the compilation stage they appear in. These annotations are called *staging annotations*. Staging annotations however are not limited to multi-stage languages. For example, *C++* [7] is a two-stage language where the first stage is the interpretation of the templates (denoted by the < > tags) and the second stage is the compilation of the non-template code.

Meta-programming can help achieve various benefits [8], the most typical of which is performance. It provides a mechanism for writing general purpose programs without suffering any overhead due to generality; rather than writing a generic but inefficient program, one writes a program generator that generates an efficient solution from a specification. Additionally, by using partial evaluation it is possible to identify and perform many computations at compile time based on a-priori information about some of the program's input, thus minimizing the runtime overhead. Another application is the reasoning about object-programs. It is possible to analyze properties of an object-program that can be used to improve performance, or provide object program validation. Finally, meta-programming can achieve code reusability at a macroscopic scale by implementing parameterized proven design practices (design patterns) and instantiating them based on the given parameters.

Context. As with normal programs, when writing meta-programs errors are bound to happen, so it is important to have the proper tools to understand the origin of the error and finally resolve it. In normal programs, there are two main error categories: compilation and execution errors. Compilation errors are generally resolved easily as compilers can identify exactly where something went wrong and why. On the other hand, execution errors involve runtime state that may be different between executions and is not directly visible to the programmer, making them harder to resolve. Fortunately, debuggers can provide the required information by allowing inspection of runtime values and call stack, tracing the program execution, and adding breakpoints, thus significantly aiding the error resolution process.

Problem. The same principles regarding errors and their management apply for meta-programs as well. However, both meta-program compilation and execution may involve code that was never part of the original program. This means that compilation errors are not that easy to deal with anymore as the error provided no longer reflects code that the programmer can see and understand. Moreover, meta-program execution errors are even harder to face since there is no actual source that can be used for debugging. It becomes obvious that error handling in meta-programs requires more sophisticated tools, without which the programmer's ability to write, understand and maintain meta-programs may be severely hindered. Clearly, compilation errors due to staging should encompass sufficient information to identify their cause while stage execution errors should be detectable using typical source-level debugging.

Contributions. In this paper, we discuss the implementation details of a meta-programming system addressing the previous issues based on:

- Generating source files for compilation stages and their outputs (original source transformations) and incorporating them into the project manager of the IDE, associated with the source being built. These files become part of the workspace and can be used for code review, error reporting and source-level debugging.
- Maintaining the chain of all source locations involved in the generation of an erroneous code segment to provide precise error reports. This chain includes the original source as well as the generated compilation stage source files and their outputs and can be easily traversed within the IDE to resolve any error.
- Mapping original source breakpoints to breakpoints for some compilation stage and using its generated source to provide compile-time source-level debugging.

We then detail the amendments required to support such functionality. Finally, we discuss the necessary contact sites between the compiler and the tool-chain.

2 Related Work

Our work targets the field of meta-programming in compiled languages and focuses on the delivery of an integrated system able to support debugging of meta-programs being executed at compile-time as well as provide precise and meaningful messages for compilation errors originating within meta-code. In this context, the topics directly relevant to our work are compile-time debugging and error reporting.

2.1 Compile-Time Debugging of Stages

C++ [7] support for meta-programming is based on its template system that is essentially a functional language interpreted at compile time [9], [10]. There are *C++* debuggers (e.g. *Microsoft Visual Studio Debugger*, *GDB*) that allow source level debugging of templates, but only in the sense of tracing the execution of the template instantiation code and matching it to the source containing the template definition. However, there is no way to debug the template interpretation during compilation. A step towards this end is *Templight* [11], a debugging framework that uses code instrumentation to produce warning messages during compilation and provide a trace of the template instantiation. Nevertheless, it is an external debugging framework not integrated into any development environment and relies on the compiler generating enough information when it meets the instrumented code. Finally, there is no programmer intervention; the system provides tracing but not interactive debugging.

D [12] is a statically typed multi-paradigm language that supports meta-programming by combining templates, compile time function execution, and string mixins (text code injected into the source). *Descent* [13] is an *Eclipse* plug-in for code written in *D* and provides an experimental compile-time debugging facility that supports simple templates and compile-time functions. However, the debugging process does not involve the normal execution engine of the language; instead it relies on a custom language interpreter for both execution and debugging functionality.

Nemerle [14] is a statically typed object oriented language, in the *Java* / *C#* vein that supports meta-programming through its macro system. *Nemerle* and its IDE, *Nemerle Studio*, provide support for debugging macro invocations during compile time. *Nemerle* macros are actually compiler plug-ins that have to be implemented in separate files and modules and are loaded during the compilation of any other file that invokes them. Since they are dynamically linked libraries with executable code, it is possible to debug them by debugging the compiler itself; when a macro is invoked, the code corresponding to its body is executed and can be typically debugged. Nevertheless, the development model posed, requiring each macro to be in a separate file and module, is restrictive and the macro debugging process is rather cumbersome.

There are a lot more compiled languages, both functional and imperative, that support meta-programming. Some examples include *MetaOCaml* [6], *Template Haskell* [15], *Dylan* [16], *Metalua* [17] and *Converge* [18]. However, none of them provide any support for debugging meta-programs during compilation.

2.2 Compile-Error Reporting

To our knowledge, most compiled languages that support meta-programming provide very limited error reporting for compilation errors originating from generated code. Typically, the error is reported directly at the generated code with no further information about its origin or the context of its occurrence. Below we examine some of the few cases that offer a more sophisticated error reporting mechanism.

C++ compilers (e.g. *Microsoft Visual Studio Debugger*, *GDB*) provide fairly descriptive messages regarding compilation errors occurring within template instantiations. Using these messages provided, the programmer may follow the instantiation chain that begins with the code of the initial instantiation that caused the error (typically user code) and ends with the code of the instantiation that actually triggered the error (probably library code). Essentially, these error messages represent the execution stack of the template interpreter. While potentially informative and able to provide accurate information to experienced programmers, template error messages are quite cryptic for average programmers and require significant effort to locate the actual error. Unfortunately, this is the common case for nontrivial meta-programs and applies especially to libraries with multiple template instantiations (e.g. *Boost* [10]).

Converge [18] provides some error reporting facilities related to meta-programming by keeping the original source, line and column information for quoted-code and retaining it at splice locations (injections into the program AST). For runtime errors, this approach works fine but is limited by the single source code location that can be associated with a given virtual machine instruction, not allowing for a complete trace of the error. For compile-time errors, *Converge* can track down the source information of the quasi-quotes and associated insertions (i.e. any AST creation) to provide a detailed message. However, it fails to provide information about the splice locations, which actually involve staging execution. This means that any error originating in generated code cannot be properly traced back to the code that actually produced it. Finally, any compile error reported is presented only with respect to the original source, thus providing no actual context regarding the temporary module (i.e. computation stage) being executed to perform the splice.

3 Meta-programming System

Our meta-programming system[1] is based on the *Delta* programming language [19] and its IDE, *Sparrow* [20]. To support meta-programming facilities, several extensions were made to the language itself as well as to its compiler and IDE.

3.1 Language Extensions

To support multi-stage meta-programming, *Delta* has been extended with staging annotations similar to the ones of *MetaOCaml* [21].

- ***Quasi-quotes*** (written <<...>>) can be inserted around almost any language element to enclose its syntactic form into an AST. This annotation provides the easiest way to create language values containing code segments.
- ***Escape*** (written *~(expr)*) can be used on an expression within quasi-quotes to escape the syntactic form and interpret the expression normally. It allows combining existing AST values in the AST being constructed by the quasi-quotes.
- ***Inline*** (written *!(expr)*) can be used on an expression to evaluate it at translation time and inject its value directly into the source code. For the injection to be valid the expression must evaluate to an AST or AST convertible value and it is performed by properly incorporating the evaluated AST into the main source AST.
- ***Execute*** (written *&stmt*) can be used to execute a statement at translation time. An addition to the original *MetaOCaml* annotations, it differs from *inline* as it does not modify the source. It is used for computations not expressible through expressions (e.g. loops) but also to generate code available only during compilation.

The following program is a simple example of compile-time meta-programming illustrating the staging annotations used in *Delta* (trivially adopted from [21]). Function *ExpandPower* creates the AST of its *x* argument being multiplied by itself *n* times, while function *MakePower* creates the AST of a specialized power function.

[1] The system is fully functional and its complete source code is available for public download through our Subversion repository `https://139.91.186.186/svn/sparrow/branches/meta` using a guest account (username: 'guest' and empty password).

```
&function ExpandPower (x, n) {
    if (n == 0) return <<1>>;
    else if (n == 1) return x;
    else return <<~x * ~(ExpandPower(x, n - 1))>>;
}
&function MakePower (n)  {
    return << (
        function (x) { return ~(ExpandPower(<<x>>, n)); }
    )>>;
}
power3 = !(MakePower(3)); //(function(x){return x*x*x;};)
std::print("2^3 = ", power3(2));
```

3.2 Compiler Extensions

The *quasi-quotes* and *escape* annotations are used to create and combine code segments and involve no staging computation on their own. Staging occurs due to the existence of the *inline* and *execute* annotations, which are therefore also referred to as *staging tags*. Essentially this means that any program containing these staging tags cannot be compiled until all of them are translated first. However, staging tags can be nested or their evaluation may introduce additional staging tags, so the whole compilation process requires multiple translation and execution stages (Fig. 1). To support this scheme, we extend the compilation process using the following steps:

1. Parse the original source program to produce the *main AST*.
2. If no staging tags exist in the main AST go to step 8.
3. Traverse the main AST collecting the nodes for the next compilation stage.
4. Assemble the collected nodes to create the *compilation stage AST*.
5. Normally compile the compilation stage AST to executable code.
6. Execute the produced code updating the main AST in the process.
7. Go to step 2.
8. Normally compile the main AST (final AST) to executable code.

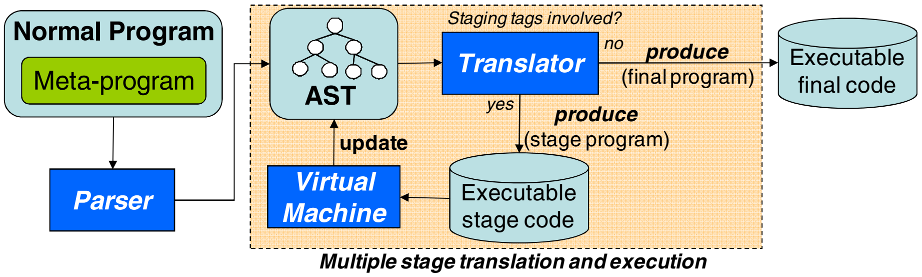


Fig. 1. High level overview of a multi-stage compilation process

Each compilation stage takes place in steps 3-6. The node collection in step 3 relies on the two following properties: (i) nested staging tags should always be evaluated at an earlier stage than outer ones; and (ii) staging tags of the same nesting level should be evaluated within the same stage. This practically means that the staging tags selected for a given compilation stage are the innermost. After assembling the nodes to create the compilation stage AST (step 4), we normally compile it to executable code (step 5). This normal compilation is possible since, by construction, the assembled AST contains no staging tags. Then we continue with the code execution (step 6), during which the original source *inline* tags that were translated to virtual machine instructions will modify the source by injecting code into the main AST. After the execution, the main AST is fully updated and ready for either the next stage (step 7), or – if no more staging tags exist – the final compilation (step 8).

3.3 IDE Support for Meta-programming

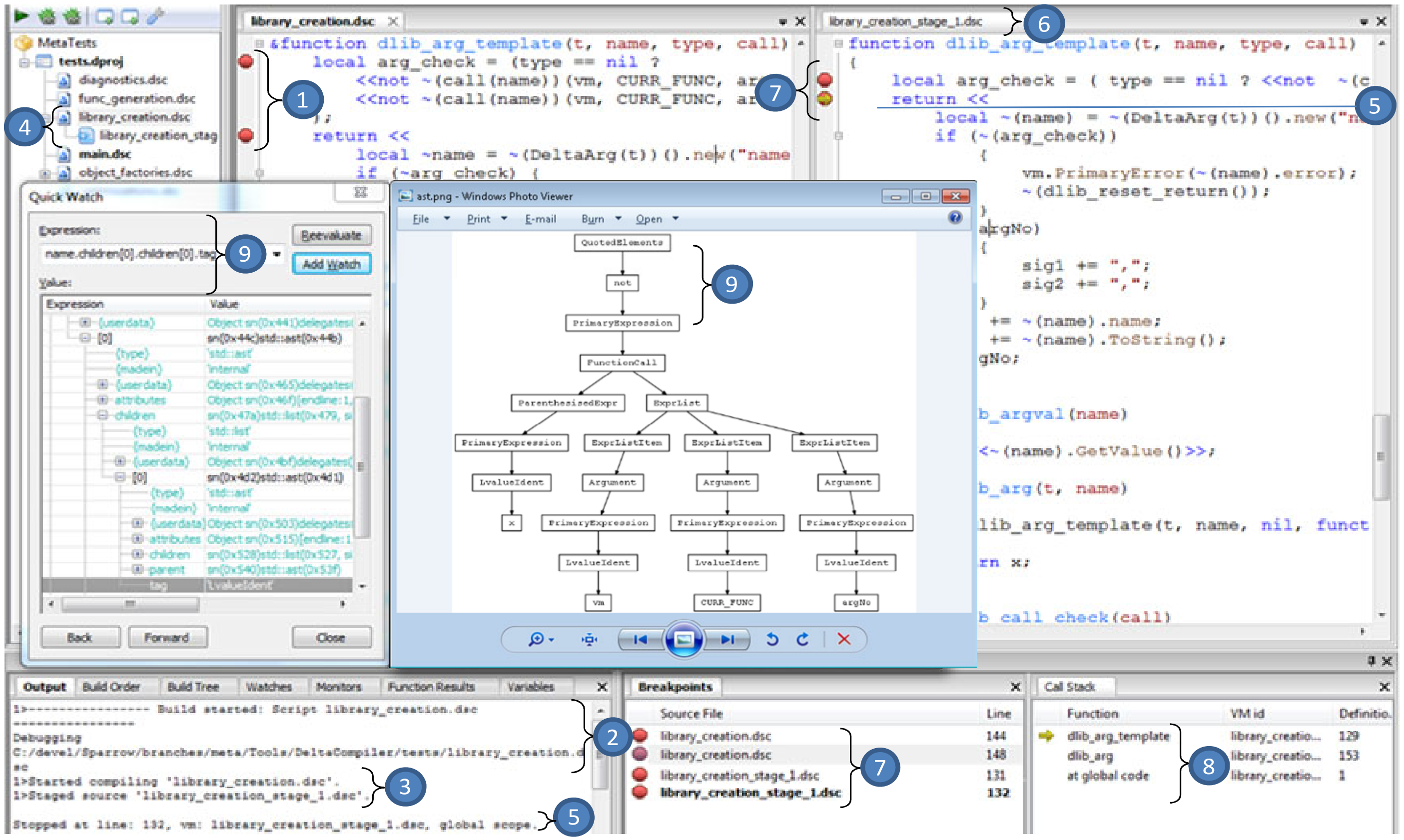
Meta-programming especially for multiple stages is a quite demanding task, so our system aims to facilitate the development of meta-programs as much as possible[2].

Since the transformations of the original source code performed by the various compilation stages are performed internally by the compiler and are therefore transparent to programmers, special attention is given to providing them with meaningful, descriptive and precise error reports in case some compilation stage raises a compilation error. Such error reports provide the full error chain across all stages involved in the generation of the erroneous code. Within *Sparrow*, programmers may easily navigate back and forth across this error chain. This feature is a significant aid in tracking the origin of the error and ultimately resolving it.

Additionally, every compilation stage is instantiated by the execution of the respective staged program. As such, it should be subject to typical source-level debugging even though its execution occurs during the compilation process. *Sparrow* provides such functionality supporting typical debugging facilities such as expression evaluation, watches, call stack, breakpoints and tracing. Fig. 2 illustrates a compile-time debugging session highlighting the following points:

1. Breakpoints are initially set within a meta-function in the original source file.
2. The source file is built with debugging enabled. This launches the compiler for the build and attaches the debugger to it for any staged program execution.
3. During compilation, the IDE is notified about any compilation stage sources.
4. Stage sources are added in the workspace associated with the source being built.
5. A breakpoint is hit, so execution is stopped at its location.
6. The source corresponding to the breakpoint hit is opened within the editor to allow further debugging operations such as tracing, variable inspection, etc.
7. The breakpoints in the generated stage source (including the one hit) were automatically generated based on the breakpoints set in the original source file.
8. The execution call stack is available for navigation across active function calls.
9. It is possible to inspect variables containing code segments as AST values.

[2] A video showing an overview of all meta-programming related features of our system is available from: `http://www.ics.forth.gr/hci/files/plang/metaprogramming.avi`



```
&function dlib_arg_template(t, name, type, call)
    local arg_check = (type == nil ?
        <<not ~(call(name))(vm, CURR_FUNC, ar
        <<not ~(call(name))(vm, CURR_FUNC, ar
    );
    return <<
        local ~name = ~(DeltaArg(t))().new("name
        if (~arg_check) {
```

```
function dlib_arg_template(t, name, type, call)
{
    local arg_check = ( type == nil ? <<not  ~(c
    return <<
        local ~(name) = ~(DeltaArg(t))().new("n
        if (~(arg_check))
        {
            vm.PrimaryError(~(name).error);
            ~(dlib_reset_return());
        }
argNo)
{
    sig1 += ",";
    sig2 += ",";
}
 += ~(name).name;
 += ~(name).ToString();
gNo;

b_argval(name)

<~(name).GetValue()>>;

b_arg(t, name)

lib_arg_template(t, name, nil, funct

rn x;

b_call_check(call)
```

Expression:

name.children[0].children[0].tag

Value:

Expression	Value
(userdata)	Object sn(0x441)delegates(
[0]	sn(0x44c)std::ast(0x44b)
(type)	'std::ast'
(madein)	'internal'
(userdata)	Object sn(0x465)delegates(
attributes	Object sn(0x46f)[endline:1,
children	sn(0x47a)std::list(0x479, s
(type)	'std::list'
(madein)	'internal'
(userdata)	Object sn(0x4bf)delegates(
[0]	sn(0x4d2)std::ast(0x4d1)
(type)	'std::ast'
(madein)	'internal'
(userdata)	Object sn(0x503)delegates(
attributes	Object sn(0x515)[endline:1
children	sn(0x528)std::list(0x527, s
parent	sn(0x540)std::ast(0x53f)
tag	'LvalueIdent'

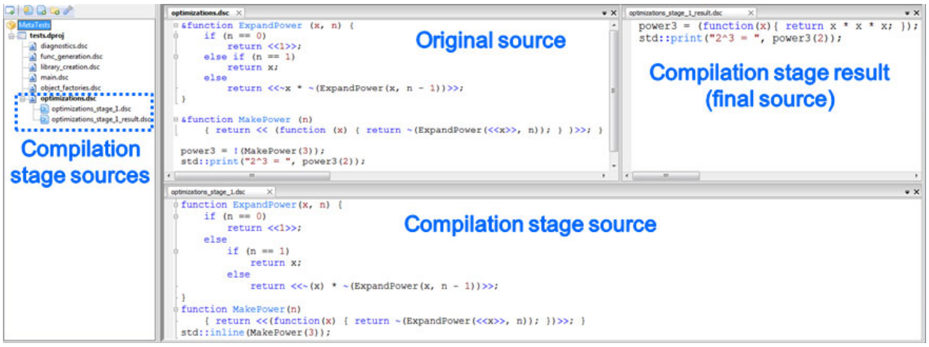
```
1>------------------ Build started: Script library_creation.dsc
-----------------
Debugging
C:/devel/Sparrow/branches/meta/Tools/DeltaCompiler/tests/library_creation.d
sc
1>Started compiling 'library_creation.dsc'.
1>Staged source 'library_creation_stage_1.dsc'.

Stopped at line: 132, vm: library_creation_stage_1.dsc, global scope.
```

	Source File	Line
●	library_creation.dsc	144
●	library_creation.dsc	148
●	library_creation_stage_1.dsc	131
●	**library_creation_stage_1.dsc**	**132**

	Function	VM id	Definitio
➡	dlib_arg_template	library_creatio...	129
	dlib_arg	library_creatio...	153
	at global code	library_creatio...	1

Fig. 2. A compile-time debugging session in Sparrow. Highlighted items 1-9 are discussed within text.

The compilation stage sources as well as their output (main AST transformation stages) are actually created and inserted into the workspace even when performing a non-debugged build. This allows programmers to review the assembled and the generated code of each stage along with the effect it has on the final program even after the build is completed, thus allowing for a better understanding of their code. Fig. 3 highlights this functionality showing all sources related to the power example.


MetaTests
tests.dproj
diagnostics.dsc
func_generation.dsc
library_creation.dsc
main.dsc
object_factories.dsc
optimizations.dsc
optimizations_stage_1.dsc
optimizations_stage_1_result.dsc
Compilation stage sources
optimizations.dsc
Original source
optimizations_stage_1_result.dsc
Compilation stage result (final source)
optimizations_stage_1.dsc
Compilation stage source

```
&function ExpandPower (x, n) {
    if (n == 0)
        return <<1>>;
    else if (n == 1)
        return x;
    else
        return <<~x * ~(ExpandPower(x, n - 1))>>;
}

&function MakePower (n)
    { return << (function (x) { return ~(ExpandPower(<<x>>, n)); } )>>; }

power3 = !(MakePower(3));
std::print("2^3 = ", power3(2));
```

```
power3 = (function(x){ return x * x * x; });
std::print("2^3 = ", power3(2));
```

```
function ExpandPower(x, n) {
    if (n == 0)
        return <<1>>;
    else
        if (n == 1)
            return x;
        else
            return <<~(x) * ~(ExpandPower(x, n - 1))>>;
}
function MakePower(n)
    { return <<(function(x) { return ~(ExpandPower(<<x>>, n)); })>>; }
std::inline(MakePower(3));
```

Fig. 3. Reviewing the compilation sources in Sparrow: Project manager view (*left*), original source file (*middle*), compilation stage source (*bottom*), compilation stage result (*middle right*)

4 Compiler Amendments

4.1 Storing the Source Code of Every Stage and Its Output

The ASTs assembled for each compilation stage are temporary and only used for code generation. To support reporting compile errors for stage sources or applying source-level debugging during their execution, these ASTs can be further utilized to create source files containing the code they represent, a process known as *unparsing*. These files are meant for programmers, so their code must span across multiple lines and be properly indented. For better visualization, we also consider that any code segment present in the initial source should keep its original form as an unparsed version may be significantly different (different indentation, empty lines, comments, etc), and the one written by the programmer is clearly user-friendlier. To support this efficiently, AST nodes contain their starting and ending character positions in the original source to retrieve their text segments (direct association of each node with its text would be far too resource demanding). This way, the unparsing algorithm will combine original and generated text segments to produce a complete source for each compilation stage.

To obtain the source code for a specific compilation stage, we apply the unparsing algorithm on its AST and store the result using some naming convention, for example adding a suffix along with the current stage number. To allow programmers to review not only the compilation stages, but also the code they generate and the modifications they perform on the main AST, we also unparse the updated main AST after the successful execution of each compilation stage. Essentially, this means that for an execution involving *n* compilation stages, there will be 2•*n* source files generated. The final program being compiled into executable code is actually the output of the last compilation stage, so it will also be available as the last generated source (Fig. 4).

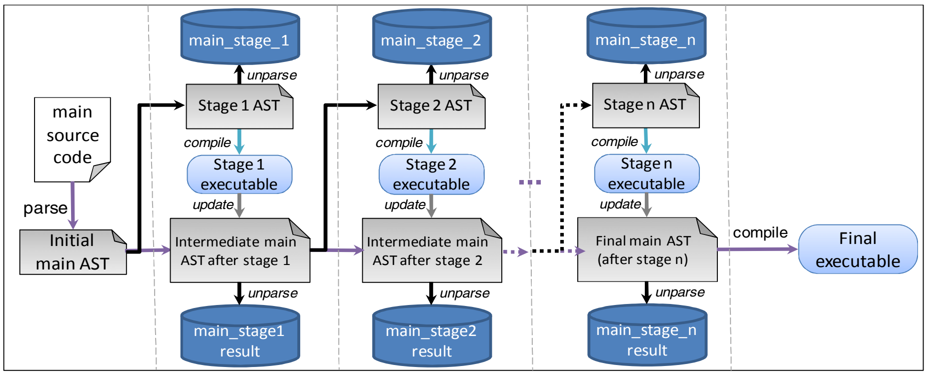


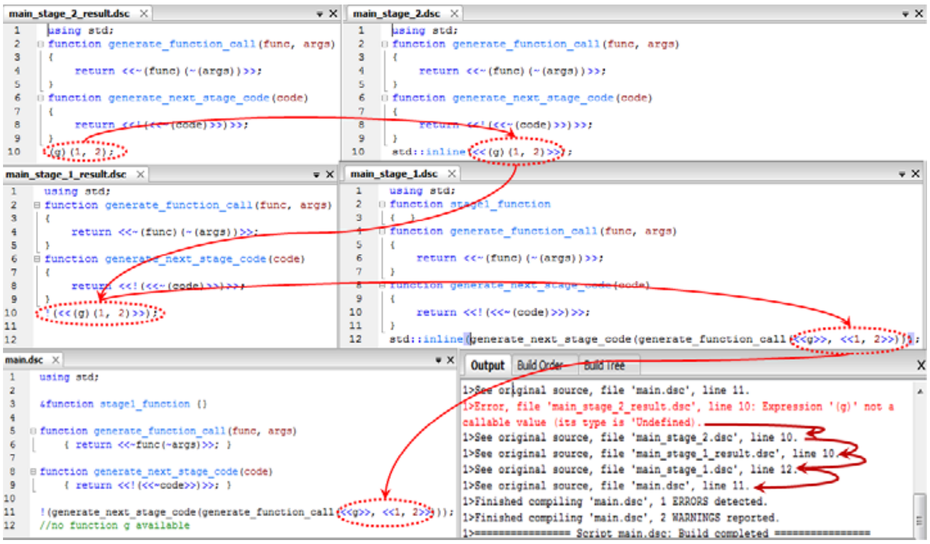
Fig. 4. Storing the source code of all compilation stages and their outputs

4.2 Tracking the Compile-Error Chain across Stages and Outputs

Any compilation stage (and the final program) is the outcome of a series of previous compilation stages and may never appear in the original source. As a result, to provide a meaningful and precise report for compile errors, the compiler has to track down the error chain across all involved stages and combine all relevant information in a descriptive message. To provide such functionality, each AST node is enriched with information about its origin, thus creating a list of associated source references. The source references for each node are created using the following rules:

1. Nodes created by the initial source parsing have no source reference.
2. When assembling nodes for a compilation stage, a source reference is created, pointing to the current source location of the node present in the main AST.
3. When updating the main AST, the source locations of the modified nodes are mapped to the latest stage source, creating the corresponding source reference.

Rules 1 and 3 along with the fact that the main AST can be modified only through the execution of the compilation stages guarantee that the main AST nodes will always either be a part of the original source or be generated by some previous stage and have a source reference to it. Furthermore, rule 2 and the fact that compilation stages are created using only nodes from the main AST guarantee the same property for all compilation stages as well. This means that any AST being compiled, either for some compilation stage or the final program, will incorporate for each of its nodes the entire trajectory of the compilation stages involved in their generation. Fig. 5 provides a sample visualization of this information upon the occurrence of an error.



```
1   using std;
2   function generate_function_call(func, args)
3   {
4       return <<~(func)(~(args))>>;
5   }
6   function generate_next_stage_code(code)
7   {
8       return <<!(<<~(code)>>)>>;
9   }
10  !(g)(1, 2);
```

```
1   using std;
2   function generate_function_call(func, args)
3   {
4       return <<~(func)(~(args))>>;
5   }
6   function generate_next_stage_code(code)
7   {
8       return <<!(<<~(code)>>)>>;
9   }
10  std::inline(<<(g)(1, 2)>>);
```

```
1   using std;
2   function generate_function_call(func, args)
3   {
4       return <<~(func)(~(args))>>;
5   }
6   function generate_next_stage_code(code)
7   {
8       return <<!(<<~(code)>>)>>;
9   }
10  !(<<(g)(1, 2)>>);
11
12
```

```
1   using std;
2   function stage1_function
3   {   }
4   function generate_function_call(func, args)
5   {
6       return <<~(func)(~(args))>>;
7   }
8   function generate_next_stage_code(code)
9   {
10      return <<!(<<~(code)>>)>>;
11  }
12  std::inline(generate_next_stage_code(generate_function_call(<<g>>, <<1, 2>>)));
```

```
1   using std;
2
3   &function stage1_function ()
4
5   function generate_function_call(func, args)
6       { return <<~func(~args)>>; }
7
8   function generate_next_stage_code(code)
9       { return <<!(<<~code>>)>>; }
10
11  !(generate_next_stage_code(generate_function_call(<<g>>, <<1, 2>>)));
12  //no function g available
```

```
1>See original source, file 'main.dsc', line 11.
1>Error, file 'main_stage_2_result.dsc', line 10: Expression '(g)' not a
callable value (its type is 'Undefined').
1>See original source, file 'main_stage_2.dsc', line 10.
1>See original source, file 'main_stage_1_result.dsc', line 10.
1>See original source, file 'main_stage_1.dsc', line 12.
1>See original source, file 'main.dsc', line 11.
1>Finished compiling 'main.dsc', 1 ERRORS detected.
1>Finished compiling 'main.dsc', 2 WARNINGS reported.
1>================ Script main.dsc: Build completed ================
```

Fig. 5. Precise error reporting for compilation stages using the entire chain of generated sources

4.3 Compile-Time Source-Level Debugging of Stages

In order to support compile-time debugging, we need to provide the source code for each compilation stage and support breakpoints both before and during debug sessions. We continue by firstly discussing the general case of debugging dynamic source code and the way we improved it when it comes to stages.

General Case: Debugging Dynamic Source Code. Debugging in the absence of a respective source file is common when either the source code is stored in a buffer or only its respective syntax tree is available for translation, usually both resulting from a computation. Clearly, this is a more general case compared to the need of source-level debugging for stages where no explicit source files are available too.

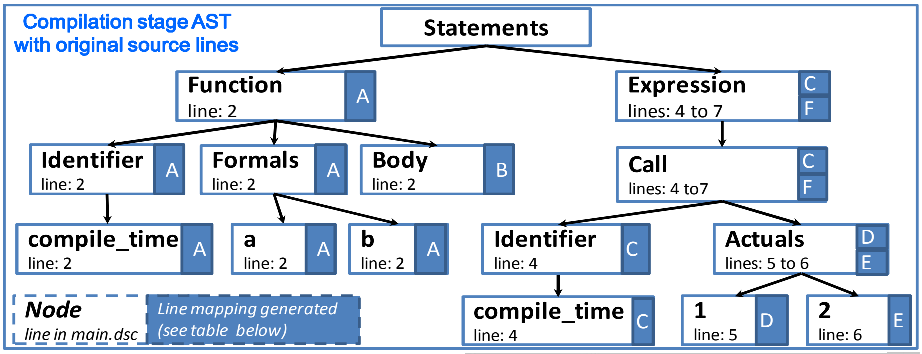
The latter is already handled in the Delta language through the reflection infrastructure as follows: The source text is incorporated into the debug information of the generated binary. Once the binary is loaded for execution, the source text from the debug information is extracted by the debugger backend and is posted to the debugger frontend when a breakpoint is hit in a statement of such dynamic source code. Then, the frontend opens an editor for the dynamic source code enabling users review it and also add or remove breakpoints as needed.

Apparently, before the initial creation of the dynamic source file, there is no way to introduce respective breakpoints. At a first glance it seems that the latter applies to stages as well, since their source code is also dynamically produced. Thus, an initial meta-compilation round is required so that the stage sources become available.

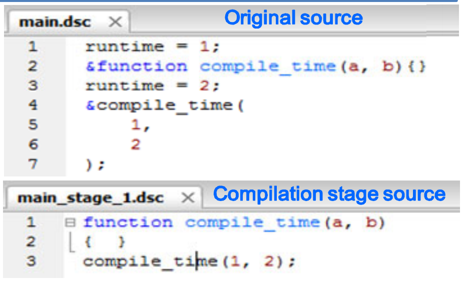
In this context, as we discuss below, we have implemented a method improving the debugging of stage source code by enabling the insertion of stage breakpoints directly on the main source file even before meta-compilation.

Specific Case: Debugging Stage Source Code. Prior to a meta-compilation round, there are no stage sources available and no breakpoints associated with their execution. The only available breakpoints concern the original source being compiled, but we can translate them to breakpoints for the dynamically generated stage sources.

As discussed, every node of the syntax tree belonging to a compilation stage can be directly traced-back across all earlier stages involved in its generation. Following this generation chain we can always reach the original source, as even nodes introduced in some particular stage will have been created recursively by code originating from the initial source. Essentially, there is a direct mapping of a compilation stage node to a node of the original source. By also keeping the reverse mapping, we can associate any node of the original source to a list of compilation stage nodes (a single node may generate multiple ones). In the same sense, we can associate each line of the original source with the compilation stage source lines that they actually generate.



	Line mapping (main.dsc → main_stage_1.dsc)
A	line 2 → line 1
B	line 2 → line 2
C	line 4 → line 3
D	line 5 → line 3
E	line 6 → line 3
F	line 7 → line 3


main.dsc Original source

```
1  runtime = 1;
2  &function compile_time(a, b){}
3  runtime = 2;
4  &compile_time(
5      1,
6      2
7  );
```

main_stage_1.dsc Compilation stage source

```
1  function compile_time(a, b)
2  { }
3  compile_time(1, 2);
```

Fig. 6. Extracting line mappings for a compilation stage: The assembled stage AST (*top*), the original source and the compilation stage source (*bottom right*) and the line mappings generated by each AST node (*next to each of the AST nodes, referring to elements of the bottom left table*)

To achieve this, we extend the unparsing process earlier discussed to associate each node line of the AST being traversed to the current line of the source being generated, taking into account the lines introduced by the unparsing implementation (Fig. 6). Finally, we can use this association to transform breakpoints intended for the original source into breakpoints for the compilation stage sources.

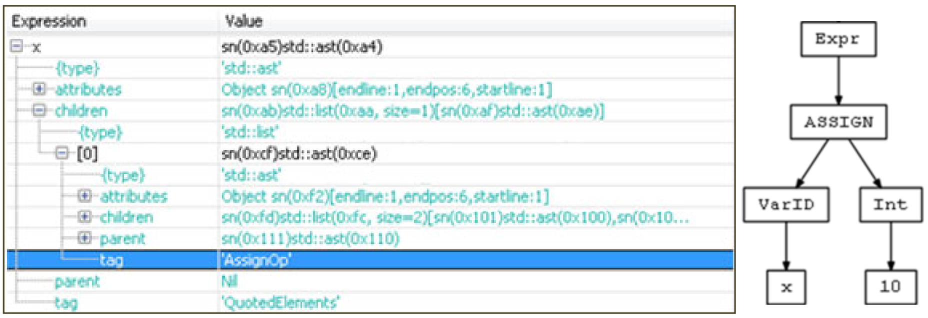
The line mappings are not unique, so a single original source breakpoint may generate multiple stage source breakpoints (e.g. a multi-line function) and multiple source breakpoints may generate the same stage source breakpoint (e.g. a complex multi-line expression that generates a single line of code). Nevertheless, this is the expected functionality supposing that code modifications occur directly at the original source line. For instance, an expression generating a function can be seen as substituting itself with a single line containing the function definition. A breakpoint set on the single line function would be hit during the execution of any statement within the function; likewise, the breakpoint of the original source will generate breakpoints for all lines the function expands to, achieving the same functionality.

5 Contact Sites between the Compiler and the Tool-Chain

5.1 Debugger

In order to support proper debugging of the compilation stages, there are two main additions required related to the debugger. The first one is regarding the expression evaluator and the need to inspect runtime values that represent code segments and the second one relates to the handling of breakpoints for the compilation stage sources.

The execution of a compilation stage typically targets the modification of the original source being compiled by adding, removing or editing code segments expressed in ASTs. In order to properly debug such operations, it should be possible to inspect such runtime values and browse through their contents. For example, the programmer should be able to inspect any specific node attribute (e.g. type, name, value, etc) as well as other related tree nodes (e.g. children, parent). The inspection facility can be delivered using typical tree views or custom tree visualization (Fig. 7).



Expression	Value
⊟ x	sn(0xa5)std::ast(0xa4)
{type}	'std::ast'
⊞ attributes	Object sn(0xa8)[endline:1,endpos:6,startline:1]
⊟ children	sn(0xab)std::list(0xaa, size=1)[sn(0xaf)std::ast(0xae)]
{type}	'std::list'
⊟ [0]	sn(0xcf)std::ast(0xce)
{type}	'std::ast'
⊞ attributes	Object sn(0xf2)[endline:1,endpos:6,startline:1]
⊞ children	sn(0xfd)std::list(0xfc, size=2)[sn(0x101)std::ast(0x100),sn(0x10...
⊞ parent	sn(0x111)std::ast(0x110)
tag	'AssignOp'
parent	Nil
tag	'QuotedElements'

Fig. 7. Inspecting code segments expressed in AST form: using an expression tree view (*left, Sparrow IDE Zen debugger*) and using a graphical tree visualizer (*right, GVEdit for Graphviz*)

Being able to specify stop points for a program execution is a vital debugging facility, so it is important to support adding breakpoints for the execution of any compilation stage. However, the compilation stage sources are dynamically produced during compilation, so it is not possible for the programmer to add breakpoints to them prior to their creation and execution. When launching the compiler, the IDE uses the debugger frontend to collect the breakpoints for the original source and passes them to the compiler to later map them to breakpoints for the compilation stage sources. During compilation, after performing the requested breakpoint mappings, the compiler has to notify the debugger backend about the new breakpoints. Instead of sending the breakpoints back to the IDE to propagate them to the debugger frontend that would in turn have to communicate them to the backend, it is much simpler and efficient to allow the execution system itself (i.e. the compiler) issue breakpoints directly to the debugger running within it (Fig. 8). The only additional requirement is the notification of the debugger frontend for the new breakpoints. Since these breakpoints are essentially transient, the frontend can simply keep track of them during the execution of each compilation stage and discard them after it is completed.

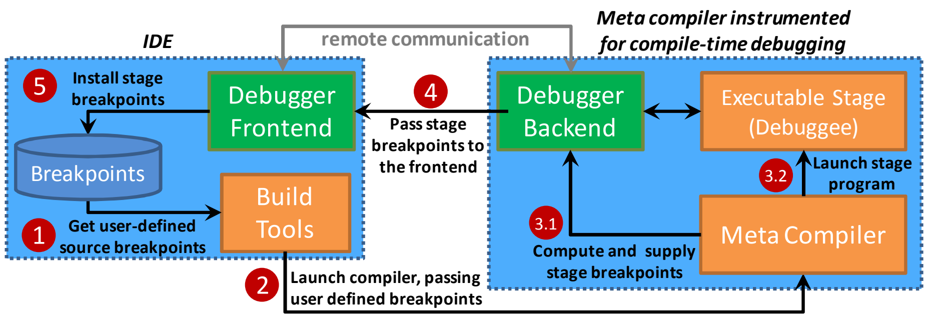

IDE
remote communication
Meta compiler instrumented for compile-time debugging
5 Install stage breakpoints
Debugger Frontend
4
Pass stage breakpoints to the frontend
Debugger Backend
Executable Stage (Debuggee)
Breakpoints
3.2 Launch stage program
Build Tools
3.1
1 Get user-defined source breakpoints
Compute and supply stage breakpoints
Meta Compiler
2 Launch compiler, passing user defined breakpoints

Fig. 8. Components involved in the generation of compilation stage breakpoints based on original source breakpoints. Arrows indicate the information flow among the components.

5.2 IDE

The compiler can be seen as a service invoked by the IDE during the build process. As such, the IDE may provide actions to be performed for specific events like compilation errors, or generation of stage sources. If the compiler is implemented as a separate executable spawned by the IDE, the communication channel between them is typically a memory pipe using standard text input and output facilities. This requires establishing a protocol for communicating the compiler events to the IDE using some text representation. For example, to notify the IDE about the existence of the stage sources, the compiler may use a special message containing resource identifiers for them (e.g. file paths). The IDE can then retrieve these files and use them to provide the sources required for the debugging process but also to maintain a reference point of internal compilation resources externalized to the programmer.

Since the IDE typically incorporates the debugger frontend, another requirement for supporting the compilation-time debugging is the ability to launch the debugger during compilation and to properly orchestrate any other facilities previously targeted only for build or debug sessions. Essentially, IDEs provide different tools during a build session (e.g. error messages, build output, etc.) and during a debug session (e.g. call stack, watches, active threads and processes, loaded modules, etc.), while usually applying different visual configurations for each activity. Compile-time debugging involves both a build and a debug session, so it is important to combine the provided facilities in a way that maintains a familiar working environment for the programmer.

6 Conclusion

In this paper we focused on providing error handling facilities in the context of meta-programming. Since meta-programs are essentially programs, proper handling is required to resolve errors occurring both during their compilation and execution. However, existing implementations provide poor error messages and lack the required support to debug meta-programs of any staging depth.

Towards this direction, we implemented a meta-programming system that features: (i) precise reports for errors occurring at compilation stages or the final program using a series of source references across the entire code generation chain; and (ii) full-fledged source-level debugging for any stage during the compilation process. To support these features, we based our implementation on the following three axes: (i) source files are generated for both compilation stages and their outputs and are incorporated into the IDE's project manager associated with the source being built; (ii) the chain of all source locations involved in generating an erroneous code segment is utilized to provide precise error reports and (iii) original source breakpoints are mapped to breakpoints for compilation stages. These features are not tightly coupled with meta-programming, so we plan to further investigate their application to other program transformation approaches like aspect-oriented programming.

To evaluate the effectiveness of our system we created a suite of meta-programs containing various errors. We then assembled two groups of programmers of similar experience and skill level and asked them to resolve the errors. One group worked with the support of our system, while the other worked without it. Results showed that the group using our system resolved the errors significantly faster. Users also noted that our debugging features allowed them to handle even complex errors quite easily.

Finally, we provided a detailed overview of the amendments required to the compilation process and tool-chain to support such functionality. We focused on an untyped language, but the same approach can also be used with typed languages. The only difference is that in our case, type errors result into stage execution errors, while a type system could detect them at the stage compilation. This however, is a typical trade-off between typed and untyped languages not related to meta-programming. All in all, apart from the meta-programming system we built for the Delta language, we believe that our work can provide a basis for extending other meta-programming systems with similar features, arguably improving the meta-programming experience.

References

1. Kernighan, B.W., Ritchie, D.M.: The C programming language. Prentice-Hall, Englewood Cliffs (1988)
2. Bawden, A.: Quasiquotation in Lisp. In: Danvy, O. (ed.) Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, pp. 88–99. University of Aarhus, Dept. of Computer Science. Invited talk (1999)

3. Martel, M., Sheard, T.: Introduction to multi-stage programming using MetaML. Technical report, OGI, Portland, OR, 211, 213 (September 1997)
4. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM), Amsterdam, pp. 203–217. ACM Press (1997)
5. Sheard, T.: Using MetaML: A Staged Programming Language. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) AFP 1996. LNCS, vol. 1129, pp. 207–239. Springer, Heidelberg (1996)
6. MetaOCaml: A compiled, type-safe multi-stage programming language (2003), `http://www.metaocaml.org/` (accessed September 13, 2011)
7. Stroustrup, B.: The C++ Programming Language Special Edition. Addison-Wesley (2000)
8. Sheard, T.: Accomplishments and Research Challenges in Meta-programming. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 2–44. Springer, Heidelberg (2001)
9. Veldhuizen, T.: Using C++ template metaprograms. C++ Report 7(4), 36–43 (1995)
10. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley Professional (2004)
11. Porkolab, Z., Mihalicza, J., Sipos, A.: Debugging C++ template metaprograms. In: Proc. of GPCE 2006, pp. 255–264. ACM (2006)
12. Alexandrescu, A.: The D Programming Language. Addison-Wesley Professional (2010)
13. Descent: An Eclipse plugin providing an IDE for the D programming language, `http://www.dsource.org/projects/descent` (accessed September 13, 2011)
14. Skalski, K., Moskal, M., Olszta, P.: Meta-programming in Nemerle (2004), `http://nemerle.org/metaprogramming.pdf` (accessed September 13, 2011)
15. Sheard, T., Jones, S.P.: Template meta-programming for Haskell. In: Proceedings of the Haskell Workshop 2002. ACM (2002)
16. Bachrach, J., Playford, K.: D-expressions: Lisp power, dylan style (1999), `http://www.ai.mit.edu/people/jrb/Projects/dexprs.pdf` (accessed September 13, 2011)
17. Fleutot, F.: Man Metalua (April 2007), `http://metalua.luaforge.net/metalua-manual.html`
18. Tratt, L.: Compile-time meta-programming in a dynamically typed OO language. In: Proceedings Dynamic Languages Symposium, pp. 49–64 (October 2005)
19. Savidis, A.: Dynamic Imperative Languages for Runtime Extensible Semantics and Polymorphic Meta-Programming. In: Guelfi, N., Savidis, A. (eds.) RISE 2005. LNCS, vol. 3943, pp. 113–128. Springer, Heidelberg (2006)
20. Savidis, A., Bourdenas, T., Georgalis, J.: An Adaptable Circular Meta-IDE for a Dynamic Programming Language. In: Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007), Luxemburg, November 26-27, pp. 99–114 (2007)
21. Taha, W.: A Gentle Introduction to Multi-stage Programming. In: Lengauer, C., Batory, D., Blum, A., Vetta, A. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 30–50. Springer, Heidelberg (2004)