The Dolorem Pattern: Growing a Language Through Compile-Time Function Execution

Simon Henniger
Technische Universität München, Germany

Nada Amin
Harvard University, Cambridge, MA, USA

Abstract

Programming languages are often designed as static, monolithic units. As a result, they are inflexible. We show a new mechanism of programming language design that allows to more flexible languages: by using compile-time function execution and metaprogramming, we implement a language mostly in itself. Our approach is usable for creating feature-rich, yet low-overhead system programming languages. We illustrate it on two systems, one that lowers to C and one that lowers to LLVM.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Language features

Keywords and phrases extensible languages, meta programming, macros, program generation, compilation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.41

Category Pearl/Brave New Idea

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.13`
Software: `https://zenodo.org/record/7720029`

Acknowledgements We thank Michael Ballantyne, Will Byrd, Anastasiya Kravchuk-Kirilyuk, and Cameron Wong for discussions about this work and feedback on drafts. We also thank anonymous reviewers for the insights and feedback.

1 Introduction

1.1 Motivation

Traditional macros serve mostly to create new syntax forms, expanding into a pre-defined core language. This means that the expressive power of macros is ultimately limited by the core language. For example, traditional macro systems could not add a "plus" macro to a language that does not have a concept of arithmetics.

This means that a language could never be fully built up from scratch with one of these macro systems. It also imposes a limit on the flexibility of macro systems: for example, they typically could not allow for supporting new features of a CPU chipset or a target language that are unsupported in the compiler.

We show a new pattern to design languages that can grow beyond their original definition. Our macros do not expand into a pre-defined core language, but rather into the target language itself. This allows us to build up a language of flexible zero- and low-cost abstractions in the form of macros.

1.2 Idea

We start with a low-level system (*the target system*), think the C environment. Using this system, we create a minimal programming language that compiles to the target system and allows for compile-time function execution. This language (which we will later call the *base language*), is barely any more high-level than the target system and just big enough to call and define functions.

We then give this language access to its own code generation functions. We also allow for certain functions defined in the language to be executed *as part of the compilation*.

As a result, we can define entirely new language features in the language and immediately use them after defining them, thereby *bootstrapping* a language from a low-level system.

This has a number of implications. First, we no longer implement a compiler as a monolithic unit. Rather, ours is an extensible system that consists of the implementation of the base language and any additional language features implemented in the base language.

Second, we effectively have heterogeneous staging from the high-level that targets the low-level system. The staging is realized by executing compiled functions at compile-time.

Our pattern provides abstractions within the high-level system to define new language features. Those are based on traditional macros, which allow adding commands to the language. In addition to macros, we allow for layers. A layer is a set of changes to the behavior of a set of existing macros, i.e. a change to the language semantics. Layers give rise to a language tower.

We show that our language provides facilities for very flexible metaprogramming and exhibits minimal overhead in many cases.

1.3 Structure

In this paper:

- We describe our design principles (Section 2).
- We describe the related concepts of heterogeneous staging and compile-time function execution (Section 3).
- Based on these design principles, we implement a small demonstration system that targets C, called dolorem-c (Section 4).
- We evaluate and demonstrate the flexibility by showing that we can implement examples (Section 5).
- Based on the result of our evaluation, we design, implement and evaluate a larger system called dolorem-llvm (Section 6), which we also evaluate (Section 7).

We discuss related work (Section 8) and conclude (Section 9).

2 Design

2.1 Definition

A language uses the *Dolorem pattern* if:

1. It provides the `lower` form to lower high-level source code to a low-level target language.
2. It provides the language user with the ability to define *macros*, which change the behavior of the `lower` form.
3. It exposes internal code generation functionality of the compiler to macros. Macro code can therefore explicitly control code generation, as though it were part of the compiler.

4. Its macros are defined in exactly the same language as any other code, rather than in a special language only available at compile-time. In particular, macros can call into regular code, allowing for compile-time function execution.
5. It is constructed through bootstrapping and staging.

We call a language a *Dolorem language* if it uses the Dolorem pattern.

Finally, we call a system that reads and interprets or compiles programs written in a Dolorem language a *Dolorem system*.

2.2 Two languages

This paper shows the design of two Dolorem languages. One of them, dolorem-c, lowers to C code, while the other lowers to LLVM IR.

2.3 Design Goals

The five parts of the Dolorem pattern definition will become primary design goals of the languages we will show. We now further elaborate on them and will then describe a set of secondary design goals that follow from the definition and help us reach our primary design goals.

2.3.1 Lower, don't evaluate

`eval` is commonly a function that takes an expression, evaluates it immediately, and returns another expression: the result of the evaluation. For example, the result of evaluating `(+ 1 1)` would be `2`, and the result of evaluating `(car '(a b c))` would be `'a`.

Dolorem systems call `lower` instead of `eval`.

`lower` takes an S-expression, parses it, lowers it into the system's target language, and returns a reference to the lowered code. The reference could be a pointer to an SSA in LLVM's Intermediate Representation, or just a string with C code. Calling `lower` on a function call, a variable name or an integer never actually calls the function, loads the variable content, or returns the value of that integer – with one exception: macros.

2.3.2 Macros are special cases of lowering

When the `lower` form is called on something that looks like a function call, it checks if the called function is marked as a macro.

If it is, it delegates to the macro which means it immediately (i.e. during compile time) calls it on its own argument and returns its result. This makes macros an example of compile-time function execution (see 3.1).

Hence, macro usages are one (and the only) instance in which the `lower` form behaves similar to the `eval` form.

Because macros have access to the entire language, the calling behavior is the only difference between macros and functions. Apart from it, they are both treated the same. This includes being compiled the same way.

2.3.3 Give macros explicit control over code generation

In Dolorem, macros can generate their own code. In order to do this, we give them explicit access to the code generation facilities in the compiler, so they can explicitly emit the target language.

2.3.4 Give macros access to the entire language

Unlike in other systems like C++'s "constexpr" functions, macros should not be restricted in which other functions they can call and what they can do.

First, this allows us to use macros in traditional metaprogramming cases, for example those where a metaprogram accesses and parses a file to use it as a basis for generating code. Second, since we use macros to directly generate backend code, it also improves the overall flexibility of the language.

Note that this requires us to consider each individual function to be its own translation unit. If we delayed compilation of a function, there could be a macro usage between the definition and compilation of it and then the function could not be used by the macro.

2.3.5 Keep the base language as small as possible

Because the Dolorem pattern uses staging, a Dolorem language is implemented in two parts: One part is implemented in some other language like C, while the rest is implemented in the Dolorem language itself (on top of the base language).

We refer to the first part as *the base language*. This part of the system should be kept minimal, i.e. just big enough to allow to bootstrap the rest of the language.

First, the Dolorem pattern aims for low overhead, so there is usually no reason not to implement something in a Dolorem language if we can.

Second, while we work to minimize it (see below), there is and will always be a barrier between a Dolorem language and its target language. So the behavior of any code that is written in a target language is a bit harder to change from within the Dolorem language.

2.4 Secondary principles

Based on this definition of the Dolorem pattern, we define a few secondary design goals that help us reach our primary goals:

2.4.1 Limit barriers

There should be as few barriers between the Dolorem language and the target environment as possible. For example, we should allow one to call the other directly, without using a *foreign function interface* or worrying too much about name mangling.

This not only makes it easier to interface Dolorem programs with existing programs written in the target environment, e.g. C (thereby increasing Dolorem's value as a systems programming language), but also allows to access the compiler's backend as seamlessly as possible.

2.4.2 Do not prescribe a model of execution

We let the language user decide about when to write code to disk/execute it rather than prescribing one model of execution. Unlike traditional language processors, Dolorem systems do not have a preferred mode of translation. Although we like to refer to them as compilers, they are neither compilers nor interpreters in the traditional sense.

Rather, a language user can decide to write and use a macro to execute code at compile-time, thereby using the Dolorem system as an interpreter, or write and use a macro to write generated code to disk, thereby using it as a compiler.

2.4.3 Layers are sets of overridden macros

We want to be able to change the behavior of macros even after their definition.

That is why we allow macros to be overridden. When we override a macro, we provide a new macro that replaces it. This new macro may call into the original macro (or an older override) as a fallback.

Since it is usually more useful to override several macros at once, Dolorem systems support layers – sets of macros that are overridden together. Layers are often used to provide new language features. For example, we will later show a "function overloading" layer that overrides the `lower` macro and the `defun` macro to allow for several functions with the same name and different numbers of arguments.

Since most macro overrides fall back on the last override, layers can stack to form a tower.

2.4.4 At global scope, everything is a macro

Similar to Lisp, there are no operators or keywords that are not also macros. This means that, at global scope, a Dolorem program is only a sequence of macros that are executed by the compiler in order of appearance.

Having nothing but macros is useful because it allows us to change every aspect of the language by overriding macros.

2.4.5 Use S-expressions

This paper is not about syntax. That said, the Dolorem pattern does place some unusual constraints on the syntax of the language it is used in:

1. The language itself is defined to be as flexible as possible and we do not want syntax to be a limiting factor.
2. We want macros to be able to read the syntax tree. Hence, the base language must provide facilities to do so. Any complexities in the syntax will therefore bloat the base language.

S-expressions are both flexible, and very easy to read and manipulate programmatically (using the famous `car`, `cdr`, `cons`), making them an ideal fit.

3 Concepts

3.1 Compile-time function execution

Compile-time function execution refers to a compiler's ability to execute functions written in its target language at compile time.

Often, this is done to compute constant expressions at compile-time. An example is C++'s `constexpr` [2].

Depending on the language, functions that can be executed at compile time may only have access to a subset of the language. In C++, for example, `constexpr` functions are relatively limited: they can only call other functions marked `constexpr` and can not access the environment.

As explained above, languages with the Dolorem pattern use compile-time function execution to bootstrap the language. Macros, the functions executed at compile time, have access to the entire language. This includes the ability to call non-macro functions (principle (2.3.4)).

3.2 Heterogeneous staging

Multi-stage programming offers a principled way of generating code by working in a language with two stages: the first stage is a program generator which when executed yields the second-stage program. Whether a program belongs to the first stage (code generator) or second stage (generated code) can be determined by syntax quotations (MetaML [10]) or types (LMS [9]). Some multi-stage programming systems like LMS (Lightweight Modular Staging) support heterogeneous staging, where the code generator and the generated code can be in different languages.

Dolorem can be seen as a form of heterogeneous staging, where the `lower` facility acts as staging. Which stage a function belongs to is determined by whether it is marked as macro. Macros, being executed during code generation, form the code generator stage, while regular functions are the second stage.

4 dolorem-c: Implementation with C Target

dolorem-c is our first attempt at an implementation of the Dolorem pattern for a system. dolorem-c is intended as a proof of concept and simple demonstration. We want to distill the essence of the Dolorem pattern, rather than present a fully viable language[1].

To keep it as simple as possible, we target C and leave out some aspects of the language for now – for example, we rely on C for the type system and do not implement any type checks ourselves.

Since the dolorem-c system has no dependencies except for a working C compiler, it is easy to try it out[2].

4.1 The `lower` macro

After reading a file, dolorem-c calls `lower` on its content. `lower` takes a list and returns generated code. In our case, this means it has the following function signature:

`struct cexp* lower(struct val* e);`, where `struct cexp*` is a type that contains a piece of generated C code and `struct val*` is an S-expression.

Any C transpiler needs to implement its own representation of C code. The primary constraint for ours is that we want the resulting interface to be as simple as possible to use. That is why we keep our representation of a value in C code to only four syntactic categories (each stored as a separate string) that we use to track whether a statement should be at global or local scope, and to be able to differentiate between expressions and full statements that precede them. These are the four strings:

- `expression`, which stores the expression itself, e.g. `a+b`. It is later inserted into a statement.
- `context`, which stores a list of statements that need to precede whatever statement `expression` will be inserted into, e.g. `int a;`. If it is non-empty, this always ends in a closing curly parenthesis or a semicolon.
- `global`, which is a list of statements on global scope that should precede whatever function this `cexp` will be inserted into.
- `header`, which is a list of headers at global scope that need to be added to the environment after `cexp` has been compiled, e.g. `#include <stdio.h>` or a `cexp` that contains `printf`.

[1] The entire dolorem-c source code can be found in this GitHub repository: `https://github.com/metareflection/dolorem-c`

[2] A version that runs in the browser can be found here: `https://shenniger.github.io/try-dolorem-c`

In the following, we will write `cexp`s as tuples of strings in square braces separated by vertical lines, e.g. `[a+sqrt(b)|||#include <math.h>]`.

For illustration, this is the behavior of `lower`:

- If `lower` is called on a number or string literal, it can directly lower that literal to C and prints that number or string literal as a `cexp` with empty local and global context (i.e. `4` evaluates to `[4|||]`).
- If `lower` is called on an identifier, it assumes it has found a reference to a variable and returns that as a `cexp` with empty local and global context (i.e. `myvariable` evaluates to `[myvariable|||]`). A more advanced Dolorem system (like dolorem-llvm which will be shown later) would try to find the variable in the current scope. dolorem-c leaves this to the C compiler.
- If `lower` finds a function call, it checks if the called function is a macro. If not, all the arguments to the call are recursively `lower`ed, and code for the call is generated. If the called function is a macro, *lower* resolves the address of the macro function, and calls it.

4.2 Macros in the base language

We want to bootstrap as much of the language as possible. In order to be able to do that, we need to add a few more macros, just enough for the user to be able to define their own functions and macros:

- `progn` calls `lower` on each of a list of expressions and returns the result of the last (similar to `progn` in Lisp).
- `include` reads another file, and calls `progn` on its content.
- `funproto` creates a function prototype for a given function signature.
- `function` creates a function (including body).
- `mark-as-macro` marks a given function name as a macro.
- `compile` lowers its argument, then compiles the resulting C code to machine code, and stores the C headers. See below for more.

Note that `mark-as-macro` and `compile` are the only two macros in this list to have side effects. The others do not modify global state at all – for example, `function` only returns C code with a function definition, but does not register that function anywhere yet.

We need `progn` and `include` in the base language because, without them, we could not read any files of source code (but only individual macro calls), and we need the others to be able to define functions and macros.

In addition to those macros, there are a few functions that help with reading and processing S-expressions like `car`, `cdr`, `val-is-string`, `expect-ident`, or `make-int-val`.

The base language contains nothing else – most notably absent are any kind of control structures, local or global variable definitions, and arithmetic expressions.

In order to reduce barriers, the base language also does not introduce any new calling conventions. One example of that is that dolorem-c's calling convention is the same as that of C and all dolorem-c symbols need to exist within the C environment. This is even true for macros: When a macro is called within `lower`, the address of the macro is being resolved by using a simple `dlsym(RTLD_DEFAULT, "macroname")`, i.e. we use the system's dynamic linker to resolve the symbol.

4.3 Marking a function as a macro

Any function that has the macro signature (`struct cexp* myfunction(struct val* e);`) can be marked as a macro. This does not change anything about the function's code or calling convention on the C side, however it has implications on how calls to the function are handled in dolorem-c code.

If `myfunction` is not marked as a macro, `(myfunction myarg)` generates code to call the function and evaluate its argument `myarg`. Since we are assuming `myfunction` to take only one argument, a `struct val*`, `myarg` would thus be expected to be a pointer to a list argument.

If `myfunction` is marked as a macro, `(myfunction myarg)` resolves the macro `myfunction` and calls it on the unevaluated S-expression form of the argument (i.e. a list of only one element: the string `myarg`).

Note that macros can be overridden, functions cannot. Thus, a call to a function is resolved by the C environment, while a call to a macro is resolved first by Dolorem's own macro table (which might say that the macro `myfunction` was overridden by the macro `myfunction2`, so that macro would be called instead).

The `lower` function has the same signature as macros, and we also mark it as a macro. This may seem counterintuitive at first, because there is no point to ever calling it as a macro (as anything in dolorem-c is evaluated by `lower` anyway). Marking `lower` as a macro is still necessary, because that way, we can override it.

It is, however, useful to be able to call `lower` as a function from within dolorem-c code. For that, we define a function `lower-now`, which acts as an adaptor and calls the current `lower` override.

4.4 Implementing base language macros

Base language macros are implemented as simple C functions. As an example for how C code interfaces with S-expressions and macros, here is `progn`:

```
struct cexp *progn(struct val *e) {
  struct cexp *r = make_cexp("", "", "", "");
  // iterate through list
  for (struct val *args = e; !is_nil(args); args = cdr(args)) {
    // call lower on each element
    struct cexp *a = call_macro("lower", car(args));
    add_cexp(r, a);
    // if not the last element, add
    // expression to context
    if (a->E && *a->E && !is_nil(cdr(args))) {
      appendString(&r->Context, print_to_mem("%s;\n", a->E));
    }
    r->E = a->E;
  }
  return r;
}
```

We begin by calling `make_cexp` to create an empty C expression. Note that the four empty strings we pass to it correspond to the four syntactic categories described above.

Then we iterate through the list we were passed, lowering each element, and adding its `context`, `global`, and `header` to our `cexp`. The `expression` of the `cexp` we return will be that of the last element of the list (because that is the return value of the block); any `expression`s of the preceding elements will be appended to `context`.

Note that we do not directly call `lower`, but rather write `call_macro("lower", ...)`. While there is a C function called `lower` that we could use, the macro `lower` might have been overridden by Dolorem code by this point, so we need to look up the current version of `lower`.

4.5 Writing functions in the base language

In the base language, simply writing `(function ...)` is not enough to define a new function. That is because `function` merely generates the `cexp` for a function, but never actually invokes a C compiler.

Actually, we need to wrap any function definition into a `compile`, for example: `(compile (function hello-world ()void (puts "hello, world!")))`

The `compile` macro first `lower`s its argument, then appends the header portion of it to a global header store, and invokes a C compiler[3] to compile it to a shared library[4]. The `compile` macro also loads this shared library.

4.6 Example macro 1: `defun`

To reduce this boilerplate, we first define `defun`, which provides us with a nicer function definition syntax and makes compilation implicit. `defun` is a good example for a simple macro that does not explicitly generate C code, but rather uses existing compiler facilities, for which it generates new input Dolorem code.

We call this kind of macro (that transforms Dolorem code to Dolorem code and then calls the `lower` macro on it for code generation) a *homogeneuous transformation* (see 4.9).

In order to define a macro, we define a function that takes an S-expression and returns a `cexp`. The dolorem-c type system cannot handle either at this early stage of bootstrapping, so the actual function signature is `void* defun(void* args);`. We compile the resulting function, and then mark it as a macro.

`defun` should be just like `function`, except that it automatically compiles for us. Since all we want is this small syntactic change, it makes sense to implement the macro as a homogeneous transformation (from Dolorem code to Dolorem code):

```
(compile (function defun ((void* args)) void*
  (call-macro "lower" (cons
      (make-ident-val "compile")
      (cons (cons (make-ident-val "function") args) (make-nil-val))))
         ))
(mark-as-macro defun)
```

To define a macro, we first create a function, then explicitly compile it, and finally mark our function as a macro. Within the function, we create a new list (using standard Lisp macros like `cons`) composed of a call to `compile` and `function` on our argument.

We then lower this list. Note that, rather than directly calling `lower` in the macro, we need to `call-macro "lower"`. We need to do so because `lower` is a macro, but in this case, we do not want to call it during code generation of `defun`, but rather want to add a call to `lower`. We could also call `lower-now` (see 4.3) and will do so in later code examples.

After `defun`, we bootstrap a `defmacro` in a similar way, such that we no longer need to write down the function signature and explicitly `mark-as-macro` every time. Note that, because of the way in which we define `defmacro`, it automatically creates a parameter called `args` that contains the list the macro was called on.

[3] Currently, both tcc and gcc have been tested. tcc is preferred on Linux because of its much faster compilation speed.

[4] We use a shared library to make sure libdl can find all new symbols, and that newly compiled code can find and access all the symbols defined before. This is why the C compiler is asked to link the new code into a shared library (with dynamic symbol lookup enabled), which is then immediately loaded. dolorem-llvm uses a more complicated JIT-based approach for this, which we will discuss later.

4.7 Example macro 2: `var`

We will now show `var` as an example of a macro that generates its own C code.

These macros, which we will refer to as *lowering transformations* lower their Dolorem input to C code.

In the case of the `var` macro, we tap into C's ability to define local variables. We want to be able to write `(var int x)` to create a new int variable called `x`. For this, we define a primitive macro called `var`:

```
(defmacro var
  (make-cexp (expect-ident (car args))
    (print-to-mem
      "%s %s;\n"
      (expect-type (car (cdr args)))
      (expect-ident (car args)))
    ""
    ""))
```

We use the newly defined `defmacro` to define the macro in only one step (rather than defining a function, compiling it, and then marking it as a macro). Within the macro's body, we call `make-cexp`, whose four arguments correspond to the four syntactic categories defined above. Two of them (`header`, `global`) are empty, while the first one is simply the name of the variable, and the second is C syntax for its definition.

The base language gives us a range of functions like `char* expect_ident(struct val* e);` and `long long expect_int(struct val* e);` that check whether a given list expression is of a certain type and, if so, unpack and return its value (otherwise, the function outputs an appropriate error).

The macro uses `print_to_mem`[5], a C function defined in the compiler that acts like `asprintf` (i.e. it formats a string and automatically allocates memory for it) to transform the dolorem-c code into C code and creates a `cexp` with the variable definition as context, only the variable name as expression, and empty global and header values.

In a similar way, we bootstrap (among others) `cond`, a Go-like `:=`, `and`, `equals`, `loop`, `add`, `sub`, and `assign`.

4.8 Language usage example: Hello, world in a loop

With the very primitive base language and our macros, we can now implement this example that prints "Hello, world!" ten times:

```
(include "def.dlr")
(defun main () void (progn
  (:= count 0)
  (while (sub count 10) (progn
    (puts "Hello, world!")
    (assign count (add count 1))))
```

We assume that the file `"def.dlr"` contains all those macros and a function prototype for `puts`.

[5] As a purely cosmetic improvement, any time an underscore appears in the C code, we can write a dash in dolorem-c. All dashes in function names are replaced by underscores when they are read.

Note that, while our program defines a `main` function, it currently does not do anything. As explained above, in order to execute code, we need to write a macro that either directly calls into the code, or writes it to disk.

We will show how to use Dolorem systems as compilers later. For now, let's execute our code directly:

```
(defmacro run (progn
  (main)
  (make-cexp "" "" "" "")))
(run)
```

We create a macro called `run` that calls into `main` and returns an empty `cexp`. Then, immediately after defining the macro, we call it.

4.9 Different types of transformations

As discussed above, there are two archetypes of Dolorem macros:

- *Homogeneous transformations*: These macros transform the S-expression they are given and then (at their very end) call `lower` to generate code for the transformed expression. Lisp can only work with this kind of macro.
- *Lowering transformations*: These macros parse the S-expression they are given and then call into the backend to generate code for it.

To demonstrate the difference, we will show two different implementations of the `quote` macro.

Most Lisps include a `quote` macro that, rather than evaluating an expression, passes its original S-expression along. Quoting is equally useful in Dolorem, but must be implemented differently. Rather than simply returning the original S-expression, we must generate code to create a given S-expression.

First, we will implement `quote` as a homogeneous transformation. We start with a function that, given a list, returns, as a list expression, a sequence of function calls to generate that list. For example, for `(1 2 3 test)`, our function should return `(cons (make-int-val 1)(cons (make-int-val 2)(cons (make-int-val 3)(cons (make-string-val "test")(make-nil-val)))))`. We call this function `lower-quote`. We can define the actual macro as simply `(defmacro quote (call-macro "lower" (lower-quote args)))` (i.e. call `lower-quote`, lower the result and return it).

`lower-quote` is defined as a series of `cond` statements[6] that call a number of simple helper functions like `make-make-funcall` or `make-cons-funcall` that generate the actual calls. This is what the statements for lists and integers look like:

```
(defmacro quote (progn
  (var res void*)
  (cond (val-is-int args)
    (assign res (cons (make-ident-val "make-int-val") (cons args (
        make-nil-val)))))
  # ...
 (call-macro "lower" res)))
```

[6] dolorem-llvm allows the implementation of a proper three-way if. In dolorem-c, this is hard due to the missing type system.

This has several advantages. First, it is relatively easy to read (and, in fact, after `quote` has been defined, such functions are even more readable). It also does not require much advanced knowledge on the inner workings of the compiler or C. Furthermore, this macro is relatively forward-compatible and will continue to work with updated code generators, although future Dolorem syntax changes may break its compatibility.

Its primary issue is that it is not particularly fast. Creating a list expression that is then immediately parsed by the `lower` macro called in the last line is wasteful. We can write a faster version of it as a lowering transformation that directly generates C code:

```
(defmacro quote (progn
  (var res char*)
  (cond (val-is-int args)
    (assign res
      (make-cexp (print-to-mem "make_int_val(%i)" (expect-int args))
         "" "" ""))
 r))
```

While it is lower overhead, this way of implementation is also much more error-prone as we have to directly manipulate strings of C code.

There is no universal rule for when to use which of the two types of transformations. Both of them have their own advantages and disadvantages: lowering transformations are faster and more flexible, but error-prone, while homogeneous transformations are simpler, yet slower.

4.10 Example macro 3: Arithmetic operators

One of the traditionally more repetitive tasks within writing a compiler is to write the code for all the arithmetic operators. It is almost always the same code for each operator, which leads to code duplication within the compiler. We will show how to leverage dolorem-c's metaprogramming abilities to help us with bootstrapping the language and avoid having to write each operator separately.

An arithmetic operator needs to read the left-hand side and the right-hand side expression, then append the context of one to the other, and finally set the expression to something like "a+b". In order to save memory, we do not create a new expression value and instead reuse one of the arguments. Here is an example for `add`:

```
(defmacro add (progn
  (:= left (lower-now (car args)))
  (:= right (lower-now (car (cdr args))))
  (add-cexp left right)
  (set-expression left (print-to-mem "(%s) + (%s)" (get-expression
     left) (get-expression right)))
 left))
```

We see that almost all of this will be the same for all operators, apart from the name of the macro and the format string (`"(%s) + (%s)"`).

Using a `quasiquote` implementation based on the `quote` macro shown above, we quote the entire macro, adding in the two changing parts via `quasiunquote`:

```
(quasiquote (defmacro (quasiunquote (car args)) (progn
  (:= left (lower-now (car args)))
  (:= right (lower-now (car (cdr args))))
  (add-cexp left right)
```

```
  (set-expression left (print-to-mem (quasiunquote (car (cdr args)))
     (get-expression left) (get-expression right)))
 left)))
```

Finally, we wrap this into a call to `lower` and add it into a macro:

```
(defmacro generate-arithmetic-operator
 (call-macro "lower" (quasiquote defmacro (quasiunquote (car args)) (
    progn
    (:= left (lower-now (car args)))
    (:= right (lower-now (car (cdr args))))
    (add-cexp left right)
    (set-expression left (print-to-mem (quasiunquote (car (cdr args))
       ) (get-expression left) (get-expression right)))
   left))))
```

We have now defined a macro that defines macros.

To use it, we write the following (at global scope):

```
(generate-arithmetic-operator add "(%s) + (%s)")
(generate-arithmetic-operator sub "(%s) - (%s)")
(generate-arithmetic-operator mul "(%s) * (%s)")
(generate-arithmetic-operator divi "(%s) / (%s)")
```

4.11 Macro Overriding

With what we have shown so far, plenty of macros can be implemented into dolorem-c. However, there is currently no way to change a macro once it is defined.

Macro overriding changes that. We add a virtual table of macros to the base language. Now, whenever a function is marked as a macro, its address is stored in the virtual table, and whenever we call a macro, we take the address from the table rather than asking the dynamic linker for it.

We also define a new function `macrofunptr override_macro(const char* name, const char* newfun);` that overrides the macro `name` by the replacement function `newfun` and returns a pointer to the old entry in the table.

With this, we can use the standard C pattern to change the behavior of existing macros:

1. We store the old function pointer from the virtual table in a global variable.
2. Then, we override the virtual table entry with a new function.
3. In the new function, we use the old pointer in the global variable as fallback.

4.12 Overriding Example: Location hints

A useful easy example is this overridding of `lower` that adds location hints to the C code so that the C compiler can emit correct debug information[7]. Any dolorem-c code compiled after a call to the function we are about to define (`load-csrchints`) can be stepped through line-by-line in GDB.

We start by creating a global variable to store the current `lower` function: `(compile ( global-var csrchints_old_lower macrofunptr))`

[7] gcc supports reading hint lines in the format of "# <filename> <line number>" to be able to tell where a piece of source code is originally from. We add these to the 'context' of the `cexp`.

After this, we define a macro called `load-csrchints` that loads the C source hints override. This is where the overloading happens. The macro is supposed to be used at global scope, so we make it return an empty `cexp`:

```
(defmacro load-csrchints (progn
  (assign csrchints-old-lower
          (override-macro "lower" "lower-csrchints"))
  (make-cexp "" "" "" "")))
```

Finally, we define the new macro `lower-csrchints` that will replace `lower`:

```
(defmacro lower-csrchints (progn
  # Call lower-level `lower`.
  (:= r (csrchints-old-lower args))
  (var filename char*)
  (var line long)
  (get-loc-info args (ptr-to filename) (ptr-to line) 0 0)
  (append-cexp r
    (get-expression r)
    (print-to-mem "# %li \"%s\"\n" line filename) "" "")
 r))
```

Note that this implementation prints duplicated hint lines whenever many subexpressions are on the same C source line. It also currently does not handle blocks correctly, often giving only the number of the first line of a multi-line block. Although it is not entirely correct, it is good enough to be useful in many debugging scenarios.

4.13 Layers

As explained above, we want to be able to override multiple macros at once and use a convenient syntax that abstracts all the details (global variables, calls to the `override-macro` function, etc).

For example, we want to be able to write `(new-layer foo (lower (new-lower-code...) ))` to override `lower` and to have all overrides, variables, and a function `load-layer-foo` automatically generated.

In dolorem-c, this higher-level layer syntax is defined as a homegeneous transformation macro, and it is not part of the base language, but rather defined in the language itself.

4.14 Layer Example: Function overloading

To demonstrate layers, we will add a very basic version of function overloading to dolorem-c.

dolorem-c does not have a type system, therefore function overloading by type is not possible. Hence, this example only overloads on the number of arguments, not the type.

To implement function overloading, we need to do two things. First, we need to change the `defun` macro such that it mangles the name of any newly created function name by adding the amount of arguments to the name. Second, we change `lower` such that it correctly resolves any overloaded function calls.

Note that one of the rather unusual features of this overloading layer is that we want mangled and non-mangled functions to coexist as seamlessly as possible. This is necessary because, otherwise, loading the layer would interfer with calling any functions defined before it was loaded. In order to achieve this goal, we add a hashmap that contains the names of any mangled functions. Whenever we find a function whose name is not in the hashmap, we do not change the call.

We will only change `defun`, not `function`. This means any functions that are defined using other means will not be mangled. This is useful because it avoids breaking macros or any other mechanism that might be defined and needs more low-level control over functions.

This is the entire source code of the layer:

```
(compile (global-var overloaded-fun void*)) # hashmap for names of
    mangled functions
(new-layer funoverload
  (init (assign overloaded-fun (hashmap-new)))
  (defun (progn
    (:= name (car args))
    (:= n-args (count-len (car (cdr args))))
    (hashmap-put overloaded-fun (expect-ident name) "")
    (val-set-string name (print-to-mem "%s__fo_%i" (expect-ident name
       ) n-args))
   (fallback args)))
  (lower (progn
    (cond (val-is-list args) (progn
      (var dummy char*)
      (cond (not
                 (hashmap-get overloaded-fun
                              (expect-ident (car args))
                              (ptr-to dummy)))
        (val-set-string
                         (car args)
                         (print-to-mem "%s__fo_%i"
                                       (expect-ident (car args))
                                       (count-len (cdr args)))))))
   (fallback args))))
```

We can now load the layer with `(load-layer-funoverload)`.

Within the layer implementation, we first add some initialization code for the hashmap, and then override the `defun` macro. Our new `defun` looks for the name of the function to be defined, saves it in the hashmap, and then mangles it by appending `"__fo_X"` (where X is the amount of parameters). Finally, it calls whichever `defun` implementation is in the tower before this layer on the newly changed definition.

We also override `lower`. If it finds a function call, our new implementation checks if the function name is in the hashmap. If it is, it mangles it based on the amount of arguments supplied. Finally, it delegates to the next layer.

4.15 Implementing optimizations

We want to be able to write code that transforms and optimizes already-generated code. To do so, we implement a transformation layer, similar in principle to the one implemented in section 4.12 (which adds location hints), except that our new layer will change the code.

As we implement our optimization, we will run into one major limitation, which is not related to the concept of the Dolorem pattern, but rather to a specific design decision we made for dolorem-c. Our representation of C code is purely textual, as opposed to using a syntax tree that is easier to change, making it a pain to read and transform once it has been generated. Using another representation for C code (such as some kind of syntax tree) would have made this easier. Unfortunately our textual representation will make our optimization more inelegant and inflexible than necessary.

In section 4.10, we added a division operator to dolorem-c. It is common for compilers to transform integer divisions of a variable and a constant to a bit shift if the constant is a power of two. We will implement this optimatization (except, since we are only showing this as an example and try to simplify as much as possible, we will only transform a division with two to a bitshift and ignore the other powers of two)[8].

If we want to transform division code, we can either (1) change the `divi` macro by overriding it in a layer, or (2) look for divisions in the code after it has been fully generated, and then change the code. Only (2) is a transformation in the narrow sense, as (1) does not technically transform generated code, but rather changes the code generator. In the Dolorem context, both have a similar effect.

We will first implement a layer for (1):

```
(new-layer bitshift-instead-of-division
  (divi (progn
    (:= first-operand (car args))
    (:= second-operand (car (cdr args)))
    (:= result NULL)
    (cond (and
     (val-is-int second-operand)
     (equals (expect-int second-operand) 2)) (progn
      (:= c (lower-now first-operand))
      (assign result (make-cexp
        (print-to-mem "(%s) >> 1" (get-expression c))
        (get-context c)
        (get-global c)
        (get-header c)))))
    (cond (not result)
      (assign result (fallback args)))
   result)))
```

This overrides the `divi` macro itself, and adds a check for whether one of the operands is the number two.

Alternatively, we can implement option (2) and search and replace existing code for any divisions with two. Due to the limitations with the C code representation, we do this on a textual level.

All divisions will be within functions, so the easiest way to catch all of them is to override the `function` macro (used within `defun` as described above) and work with its output:

```
(new-layer bitshift-instead-of-divisions-2
  (function (progn
    (:= code (fallback args))
    (:= global-ptr (get-global code))
    (:= pointer global-ptr)
    (:= modified 0)
    (loop (assign pointer (strstr pointer "/ (2)")) (progn
      (memcpy pointer ">>  1" 5)
      (assign modified 1)))
    (cond modified
      (set-global code global-ptr))
   code)))
```

[8] Since all code is compiled (and optimized) by a C compiler which will most certainly do this optimization, our code does not actually yield a speed boost, but merely serves to illustrate how code transformations are implemented with the Dolorem pattern.

First, we call the overridden macro to generate the function code and then we search the `global` of its result (which is where the function body will be).

Note that this second implementation does not refer to the `divi` macro at all. In fact, it does not matter whether the division was created by that macro, some other macro, or even a combination of several macros. All that matters is that the generated code eventually shows up in a function, and once it does show up in the function, we have access to all other code in the function and can use this contextual information for our optimization. That is how we can also implement more complex optimizations that touch multiple macros.

As we discussed, the transformations themselves are implemented in a rather inelegant way, but still, this experiment shows the principles of how source code transformations can be implemented in Dolorem:

1. For simple optimizations, it often makes sense to change how the code is being created, as opposed to transforming it after it was already created.
2. To transform code, create a layer around a macro like `function` (or `compile`) that all code will pass through, and modify it there. Then, our optimization can even touch multiple macros and operate on an entire function (or even several functions).
3. The specific range of practical transformations depends on the design of the data structure generated code is stored in. The Dolorem pattern itself is not a limiting factor.
4. If an optimization is impractical to do, that does not mean it is impossible entirely. Dolorem systems use a target language and target compiler outside of Dolorem. Sometimes, it can make sense to optimize there.

More experimentation is needed to see whether this approach scales to more complex optimizations.

5 Discussion of the C implementation

We have successfully implemented a Dolorem system that targets C.

5.1 Code size

Minus header files, the hashmap implementation, the driver and the reader/parser, the entire compiler only has 377 lines of code. This shows that we were able to keep the base language very small. In an additional 284 lines of Dolorem code, we were able to bootstrap a relatively usable language. This confirms our original plan to keep the base language minimal and bootstrap the rest.

5.2 Scope

We have left out some important aspects of a language, most importantly the type system. This is certainly a major reason for why the implementation is so small in terms of code size. In the rest of the paper, we will show that the Dolorem pattern can scale and work for a more complicated language.

5.3 Generated code

When we output C code generated by dolorem-c and change the formatting a bit, it looks very similar to human-written C code (see Figure 1). Of course, this depends on the macros used. Macros may introduce additional complexities, and thus, additional overhead, if the abstraction they provide is costly or if they are badly written and introduce unnecessary cost – but none of the macros we have shown or used in examples do.

Figure 1 *left:* dolorem-c code for a non-recursive Fibonacci function, *middle:* generated C code (indentations added for readability), *right:* hand-written C code.

```
(defun
 fib ((int x)) int
 (progn
  (var i int)
  (var n1 int)
  (var n2 int)
  (var n3 int)
  (loop
   (less i x)
   (progn
    (assign n3 (add
        n1 n2))
    (assign n1 n2)
    (assign n2 n3)
    (assign i (add i
         1))
  ))
  n3
))
```

```
int fib__fo_1(int x) {
 int i = 2;
 i;
 int n1 = 0;
 n1;
 int n2 = 1;
 n2;
 int n3 = 0;
 n3;
 while((i) < (x)) {
  (n3) = ((n1) + (n2));
  (n1) = (n2);
  (n2) = (n3);
  (i) = ((i) + (1));
 }
 return n3;
}
```

```
int fib(int x) {
  int i = 2;
  int n1 = 0;
  int n2 = 1;
  int n3 = 0;
  while (i < x) {
   n3 = (n1 + n2);
   n1 = n2;
   n2 = n3;
   ++i;
  }
  return n3;
}
```

This also means we can compile it with a traditional C compiler, and the binary will be basically indistinguishable from that created from an equivalent program originally written in C.

5.4 Run-time overhead

The above already gives us a first indication that dolorem-c has no or very low overhead. Since the implementation is more of a proof-of-concept, we have not measured its performance, but have rather, in many cases including the one shown in Figure 1, seen that its C code is essentially equivalent to what we would have written by hand. We will later show a more detailed run-time analysis for dolorem-llvm.

5.5 Compile speed

Since dolorem-c can be used to directly execute code (rather than as a transpiler which code is exported from), compile speed is essential.

We expect most of the overall compile time to be spent in the C compiler. In order to verify, we implement a command line flag (`-M`) that instructs the compiler to measure the overall run-time and the time spent waiting for calls to the compiler, and output a percentage.

We test this while compiling several programs examples, including the definitions of the macros presented here (and more). None of our test cases does anything, they all compile and then terminate.

As expected, measurements show that 95-97 % (tcc) or more than 99 %[9] of overall compile time is spent in the C compiler. The discrepancy is due to the fact that gcc is about 20 times slower than tcc, likely because its code generation is generally slower and also because it is not optimized to be invoked for each function separately.

Subtracting the time spent in the C compiler, dolorem-c takes between 5 and 10 milliseconds to compile the entire collection of standard macros (around 500 lines of dolorem-c code)[10].

These measurements show that, while the dolorem-c compiler is comparatively fast given its flexibility, C compilation drastically slows it down. To improve speed, the implementation should implement some form of caching for the compilation. We will later show an example of this for dolorem-llvm.

6 Implementation with LLVM Target

We are now ready to implement the Dolorem pattern in an LLVM-based [7] system. While dolorem-c was intended as a proof of concept and simple demonstration, dolorem-llvm is intended to include more complex features like static typing, even if this makes it more complex[11].

The design of dolorem-llvm is similar to dolorem-c in many ways. We will not provide a full outline of the design, but rather describe the differences to dolorem-c, with a focus on providing some general discussion on how to apply the Dolorem pattern to a more complete compiler system.

Rather than generating code as a string, dolorem-llvm calls into LLVM directly to build LLVM IR.

LLVM's main interface is template-heavy C++ code. Since interfacing with that would drastically increase the size of the base language, we instead mostly interface with the C bindings (llvm-c).

6.1 The `lower` macro

Obviously, the most important difference is that there are no `cexp`s anymore, because the new system does not translate to C.

Instead, `lower` returns a pointer to an `rtv` (*run-time value*) rather than a pointer to a `cexp`. An `rtv` wraps an `LLVMValue`, i.e. a reference to a value computed by a program. It also contains the type information the new type system needs.

6.2 Functions

`LLVMValue` works differently from a `cexp` in one key way: Because a dolorem-llvm `rtv` can hold only a value within a function, rather than an arbitrary piece of code, a dolorem-llvm macro can not return a full uncompiled function.

[9] The results are very similar for all test cases and, while there is quite some variance between runs, it seems random and unrelated to the length or any other property of the test case. That is why we do not provide a detailed list of measurement values.

[10] Measured on an Intel Xeon E3-1505M v5, when compiling the base language with gcc 12.2.

[11] The entire dolorem-llvm source code can be found in this GitHub repository: `https://github.com/shenniger/dolorem`

Therefore, the `function` macro can not be implemented like in dolorem-c, and dolorem-llvm does not define one, but rather implements `defun`, and `defmacro` in its base language. There are separate macros to define lambdas.

In dolorem-c, the `defun` macro is essentially composed of two different macros, namely the `function` macro which generates C code for the function, and the `compile` macro which hands the generated code to a C compiler.

This shows that, while dolorem-llvm and dolorem-c both try to keep their base language minimal (principle (2.3.5)), LLVM's more complex API often causes macros to be moved into the base language.

6.3 Language usage: Compiling code

Above, we already showed how to execute Dolorem code directly during compile-time (see 4.8). However, principle (2.4.2) says that it should also be possible to compile Dolorem code.

In order to do that, we need to write LLVM IR code to disk, and then compile it:

1. LLVM only supports to write modules to disk, so by directly calling LLVM's `LLVMModuleCreateWithName`, we create a new LLVM module.
2. For each single function that is to be written to disk, we call dolorem-llvm's `copy-symbol-to-module`. This function looks up a symbol in an array that dolorem-llvm copies all function modules to before they are JIT-compiled and copies it to the new module. Note that this works because, like dolorem-c, dolorem-llvm compiles each function separately and in its own `LLVMModule` (see principle (2.3.4)).
3. Call an LLVM function that exports a module, e.g. `LLVMWriteBitcodeToFile`.
4. Finally, we compile this module to machine code. The easiest way to do this is to invoke `clang` on the command line, but this could also be done using the LLVM library.

Here is a simple example that compiles a "Hello, world!" program to a Linux executable[12]:

```
(defun main () i32 (progn
  (puts "Hello, world!")
 0))
(defmacro compile (progn
  (:= mod (LLVMModuleCreateWithName "test"))
  (copy_symbol_to_module "main" mod)
  (LLVMWriteBitcodeToFile mod "tmp.bc")
  (system "clang tmp.bc -o hello-world";
 (empty-rtv)
))
(compile)
```

A program compiled in this way does not depend on the compiler in any way; it is a freestanding executable, quite similar to an executable a C compiler would have created for the same program.

6.4 Precompilation

We can use this mechanism for precompilation.

We compile all functions of a file (including macros) to a shared object. Next time we compile the same file, we still read it entirely and execute all macros, but do not compile any

[12] `(empty-rtv)`, like dolorem-c's `(make-cexp "" "" "" "")`, creates an empty macro return value.

functions within the file if it is unchanged. Instead, we simply load the shared object from last time. This way, we continue to populate all internal data structures like function lists, but avoid calling into LLVM, which is the most time-consuming step.

We would typically use this on macros that define the language. Not having to recompile them every time we compile any dolorem-llvm code reduces compilation time.

This mechanism is important because it means that, unlike other staged languages, dolorem-llvm does not need to recompile the entire language with every translation unit.

6.5 Optimizations

In section 4.15, we discussed the implementation of code transformations and optimizations in the Dolorem system. We also already explained four basic principles of Dolorem source code transformations.

We will now attempt to use LLVM to optimize dolorem-llvm code. To do so, we use the four principles. Our attempt here differs from the attempt in section 4.15 in two main respects: First, since dolorem-llvm does not generate code as text, but instead constructs LLVM's data structure which is powerful and easy to modify, we will modify this data structure rather than text. Second, LLVM provides a number of facilities to implement and run optimizations which we can access because we have full access to LLVM (as part of 2.4.1).

That is why, unlike in dolorem-c, we will not implement our own layer for optimizing a function, but will instead show how to apply LLVM's `PassManager` to a dolorem-llvm function. To do so, we add a macro `optimize` that first lowers its argument, then calls into LLVM to run an optimization[13]:

```
(defmacro optimize (progn
  (:= result (lower (car args)))
  (LLVMBuildRetVoid bldr)
  (LLVMRunPasses mod # the current module
                 "instcombine" # the pass name
                 (GetThisMachineStdTargetRef) # a convenience function
                     defined before
                 (LLVMCreatePassBuilderOptions))
 result))
```

Note that we do not directly change the result of `(lower (car args))` before we return it. That is because it merely contains a pointer to an `LLVMValue` of the function return value (and the pointer will not change as part of the optimizations), while the actual sequence of commands is saved in the current `LLVMModule` as a side effect of lowering. This current `LLVMModule` is exposed to macros by the compiler through global variable `mod` and is guaranteed to only contain the currently compiled function (and any nested functions).

The idea of `optimize` is that we can wrap any function body in it to optimize that function, for example:

```
defun do-some-math () void (optimize (progn
  (:= i 10)
  (printf "10 * 16 = %i\n" (* i 16))
));
```

[13] At the time when `optimize` is called, the final `ret` instruction is likely not yet generated, so we add it to the function before optimization by calling `LLVMBuildRetVoid`.

Since we chose `"instcombine"` above, LLVM will optimize arithmetic instructions, and thus (similar to the toy optimization we implemented for dolorem-c, but much more powerful) transform our multiplication into a bit shift.

Typically, optimization passes would be added by a flag or some other global compiler setting. Our solution is different: It is only applied to the specific function we want it applied on and also allows us more flexible control in other ways. Finally, our solution allows us to implement our own LLVM optimizer pass and then apply it using the method shown above. We do not show a full example for how, because there is nothing Dolorem-specific about the implementation. It is enough to write the LLVM optimizer function as a normal Dolorem function, then register it in LLVM's `PassManager` in the macro and use it (as shown above). The macro will be able to see the function definition because of principle 2.3.4 and get a pointer to the function, which LLVM can then call.

7 Discussion of the LLVM implementation

We will now evaluate four key factors of our implementation: more complex/powerful language, low code size, minimal/no run-time overhead, and fast compilation times.

7.1 More powerful language

dolorem-llvm is a more powerful and complete language than dolorem-c. It includes its own type system, can interface with C and enables a macro writer to access the full power of LLVM.

To show that we can implement nontrivial example programs in dolorem-llvm, we implemented a graphical Pong. The 192-line-long program uses the game library SDL2 to render graphics and receive inputs.

7.2 Code size

Using our own type system rather than piggybacking on C and emitting the more complex LLVM IR requires slightly more complexity in dolorem-llvm.

Minus header files, the hashmap implementation, the driver and the reader/parser, the entire compiler has about 1980 lines of code. Most of the increase compared to dolorem-c is due to the bigger base language, and also due to the higher complexity of LLVM code.

This shows that, while more complex targets do increase the size of the base language, the basic principle of bootstrapping most of the language still works.

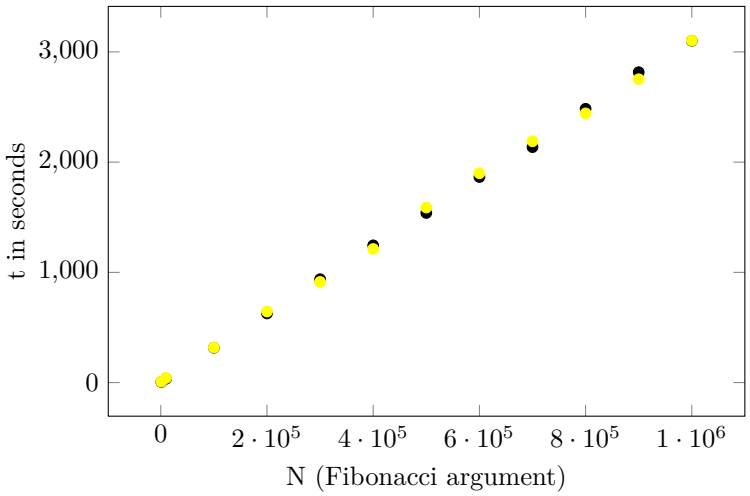
7.3 Run-time overhead

Dolorem's design attempts to impose minimal inherent run-time overhead on programs.

It is challenging to evaluate this goal in general. Primarily, that is because the performance of a Dolorem program depends on the macros used to compile it more than on anything else. In C++, if someone writes a template that is wasteful, that leads to a slower program – just like a Dolorem macro that is badly written or implements a highly ambitious and high-overhead language feature will lead to a slower program.

While it is possible to write programs without complex templates in C++, it is not possible to write non-trivial Dolorem programs without using many macros (that is because we keep the base language minimal, see 2.3.5).

Figure 2 Performance of a Fibonacci function (executed 10,000 times, milliseconds) implemented in C (black) and dolorem-llvm (yellow).



Therefore, in this section, we do not show that all Dolorem environments will be zero-overhead. We consider this to be not only impossible to show given the level afforded to macros, but also to be an undesirable goal: sometimes, using a macro that imposes a little bit of overhead might be a desirable choice, and that is fine, as long as it is a conscious, voluntary decision on the part of the developer.

Instead, we will show that, for our test cases, dolorem-llvm imposes no *inherent* run-time overhead, i.e. that dolorem-llvm allows the creation of a zero-overhead environment using macros and that it is possible to write non-trivial programs in it.

We will show this for two programs written using the default set of macros (i.e. the set of macros we have talked about in this paper). As part of our benchmark, we compile the generated code for a fibonacci function and compare it to an equivalent C implementation, using maximum LLVM optimization for both languages, i.e. `-O3` for C.

We see that the speed is exactly the same (Figure 2).

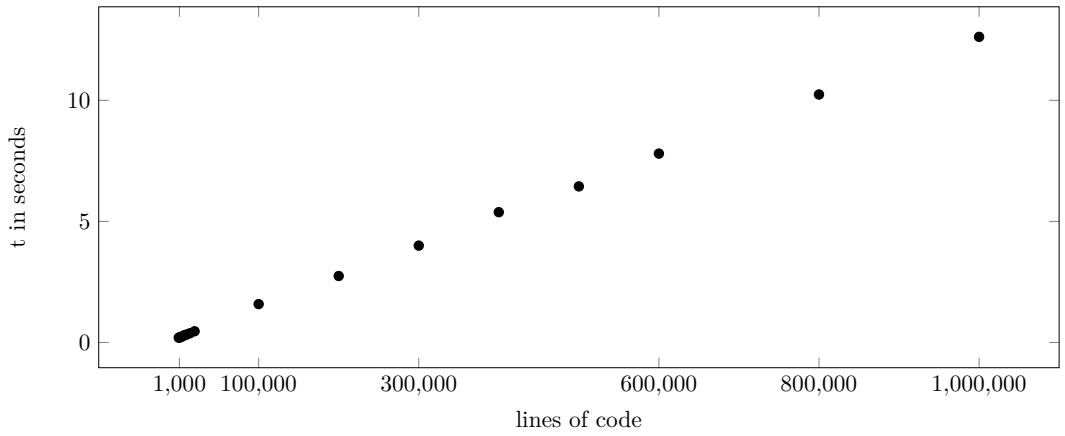
Another test shows the same result: When we compile the graphical Pong we implemented using the mechanism above and measure against an identical program written in C, we see that both are equally fast (in terms of frames per second).

Therefore, we conclude that, while additional tests are necessary to check this promise in the general case, we see that Dolorem exhibits no run-time overhead on the generated code in the cases we tested.

7.4 Compile speed

Many staged languages suffer from compile time problems, because the entire language needs to be recompiled with every translation unit. Dolorem has a unique strength that becomes relevant here: All language macros (like `if`, `add` etc.), after having been compiled, become part of the Dolorem environment, and, like any part of the Dolorem environment, can not only be JIT-compiled, but also exported as machine code (see "Do not prescribe a model of execution", section 2.4.2). Therefore, unlike in other staged programming languages, it is trivial to precompile any part of the environment (see section 6.4) – including any macros that define the language. Once precompiled, all of those language macros are only a single `dlopen` away.

Figure 3 Performance of dolorem-llvm's compiler (seconds) for a program of size N lines of code.



That is why we do not include benchmarks on loading the basic environment: basic language macros only need to be compiled once when Dolorem is first installed and then the precompiled versions can be used.

However, we are still interested in the speed of dolorem-llvm's compiler for new user code.

In order to benchmark this, we need a way of creating large amounts of code that is similar in structure to real-world dolorem-llvm code. We write a macro that duplicates existing code from a number of sources (including the standard library, the Pong implementation and a few other smaller test cases) while changing all names to avoid name clashes.

This way, we simulate how fast dolorem-llvm compiles large, real-world programs of a certain size (Figure 3).

In the case of the Pong example (which contains about 500 lines of Dolorem code in total), the compilation takes about 300 ms. Note that, because we do not use precompilation for this experiment, much of this is LLVM's start-up time, so, unlike with dolorem-c, total start-up time does not scale proportionally to the amount of code.

We can see from Figure 3 that compilation/start-up times scale linearly with code size. Taking into account that general initialization takes about 200ms, every additional 1000 lines of code take about 12ms. This is competitive with modern C++ compilers.

Precompilation massively helps additionally reduce compile speed. With precompilation enabled, even mid-size examples like Pong start instantly (< 50 ms start-up time) in direct execution mode.

8 Related Work

We discuss various approaches to achieving extensible languages. Dolorem is distinct in that it is a heterogeneous metaprogramming system which gives its macros access to the entire language. Other approaches differ:

Ziggurat [3] is a language tower system to use Scheme-like macros for languages like C. While Ziggurat has a similar aim, namely to add layer-based metaprogramming to low-level languages, Ziggurat's macros are different: they don't have access to calling functions, and act more like a preprocessor, limited to generating code. In contrast, Dolorem adheres to the principle (2.3.4) that each macro has access to the full language: a macro can call into functions, for example.

McMicMac [6] is a macro-based system for generative programming in Scheme. It provides a standard way for writing syntax transformations. Dolorem is about expanding the scope of what macros can do, while McMicMac is about putting macros to use.

Converge [12] is a system to embed DSLs in a programming language with a compile-time meta-programming facility. Compared to Dolorem, the setting is homogeneous. The system can guarantee that these embedding are safe and hygienic (in the sense of LISP macros [5]).

Like Dolorem, MetaOCaml [4] allows a developer to ask the run-time system to compile a piece of code, and link it back to the running program. However, unlike Dolorem, it is a homogeneous system. In practice, MetaOCaml can be seen as a two-stage language, where code is manually marked by quotations to ask the runtime system to compile it. The markings are not essential: inside and outside the markings, code is in the same language. They could even technically be erased yielding the same meaning with a different performance. In contrast, Dolorem is more of a metaprogramming system, where not only what is compiled can be controlled from within the language, but also *how* it is compiled.

Terra [1] and Dolorem share their aim to allow for generative low-level programming using staging. To do so, Terra is staged from within the popular scripting language Lua. Users can generate Terra code from within Lua. Terra and Dolorem also have similar mechanisms to export finished binaries programmatically. However, unlike Terra, Dolorem uses only one language and is mostly bootstrapped from within itself. Most importantly, this means that Dolorem macros have access to the entire language, while Terra and Lua are still separate, although they share the same lexical environment.

Racket [11], like Dolorem, allows macros to change the semantics of the language. Unlike Dolorem macros, Racket macros translate to standard Racket rather than directly generating code.

Metaphor [8] is an object-oriented multi-staging system that targets the .NET Runtime. Metaphor's primary design goal is to allow for type safety. While Metaphor's higher order functions have access to much of the .NET API, they mostly use it for reflection, rather than explicitly generating code.

9 Conclusion

We have described the Dolorem pattern for very flexible metaprogramming and have seen that it exhibits low overhead. Furthermore, we have shown and implemented two languages that make use of the pattern, one that lowers to C and one that lowers to LLVM.

We have discussed the design constraints that the pattern imposes on a language and presented a number of considerations, such as using S-expressions, allowing macros to call into the code generator, and compiling each function separately. We have explained the importance of the `lower` macro and how it compares to traditional `eval`-based approaches.

While dolorem-c mostly served as a proof of concept of the pattern, it could already show that our pattern can deliver on its promises of allowing easy language extensions. Its metaprogramming facilities were shown to be powerful enough to implement all arithmetic operators in less than twenty lines of code.

With dolorem-llvm, we have shown that the Dolorem pattern can work on a larger scale without losing its core advantages. To show that dolorem-llvm is fast and can interface well with existing libraries, we implemented a graphical Pong game.

Next steps

This paper showed a number of macro examples, but most of them were bootstrapping relatively standard language features. It would be interesting to create more complex layers and macros that leverage Dolorem systems to implement more advanced functionality, like instrumentation, data serialization, and borrow checking.

While we have seen in our experiments that Dolorem systems can exhibit low overhead, it would be interesting to conduct more and more methodical experiments on performance, especially those that help better understand how design choices made in macros impact overhead.

It could also be insightful to implement a Dolorem language that targets a more low-level language. For example, there could be a dolorem-x86 that translates to x86 machine code (or x86 assembly). That would create a new use-case for macros: adding support for new instruction set extensions. It would also invite more experimentation with implementing custom optimization passes as macros.

Furthermore, an interesting topic of future research is to implement cross-compilation within a Dolorem system. dolorem-c particularly could be extended by a `cross-compile` form that acts similar to `compile`, except that it compiles for a different architecture. It might also be possible to stage an entirely new language from within a Dolorem language, e.g. to compile OpenGL shaders.

It should also be investigated whether compile speeds can be further improved by changing how the C compiler is invoked or by optimizing LLVM's JIT compiler.

Finally, neither dolorem-c and dolorem-llvm currently have a clearly defined memory model convention. There should be a design contract on where memory is allocated and freed within the compiler (and, by extension, within macros). By creating a design contract, more optimizations and more safety checks would be made possible.

References

1 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. *SIGPLAN Not.*, 48(6):105–116, June 2013. doi:10.1145/2499370.2462166.

2 Gabriel Dos Reis and Bjarne Stroustrup. General constant expressions for system programming languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2131–2136, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1774088.1774537.

3 David Fisher and Olin Shivers. Building language towers with ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, September 2008. doi:10.1017/S0956796808006928.

4 Oleg Kiselyov. The design and implementation of ber metaocaml. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 2014. Springer International Publishing.

5 Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/319838.319859.

6 Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Generative and Component-Based Software Engineering*, pages 105–120, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

7 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

8 Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, pages 168–185, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

9 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, June 2012. doi:10.1145/2184319.2184345.

10 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1-2):211–242, October 2000. doi:10.1016/S0304-3975(00)00053-0.

11 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *SIGPLAN Not.*, 46(6):132–141, June 2011. doi:10.1145/1993316.1993514.

12 Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6), October 2008. doi:10.1145/1391956.1391958.