# Specialization through Dynamic Staging

Piotr Danilewski[1,2] Marcel Köster[1] Roland Leißa[1]
Richard Membarth[1,2,3] Philipp Slusallek[1,2,3]

[1]Saarland University, Germany [2]Intel Visual Computing Institute, Germany
[3]Deutsches Forschungszentrum für Künstliche Intelligenz, Germany
{danilewski,membarth,slusallek}@cg.uni-saarland.de {leissa,koester}@cdl.uni-saarland.de

## Abstract

Partial evaluation allows for specialization of program fragments. This can be realized by *staging*, where one fragment is executed earlier than its surrounding code. However, taking advantage of these capabilities is often a cumbersome endeavor.

In this paper, we present a new metaprogramming concept using staging parameters that are first-class citizen entities and define the order of execution of the program. Staging parameters can be used to define MetaML-like quotations, but can also allow stages to be created and resolved dynamically. The programmer can write generic, polyvariant code which can be reused in the context of different stages. We demonstrate how our approach can be used to define and apply domain-specific optimizations. Our implementation of the proposed metaprogramming concept generates code which is on a par with templated C++ code in terms of execution time.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Partial evaluation; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

***General Terms*** Languages, performance

***Keywords*** Dynamic staging, partial evaluation, code specialization

## 1. Introduction

The bread and butter of an optimizing compiler is partial evaluation: Parts of the program are evaluated at compile time in order to speed up the execution at run time. This includes constant propagation, loop unrolling, loop peeling, or inlining.

Due to the halting problem, a compiler cannot in general determine whether aggressive evaluation of code during compile time will terminate. Compilers therefore often cannot perform such aggressive optimization. For this reason, it is attractive to shift the responsibility to the programmer. By integrating annotations into a language one can explicitly trigger partial evaluation. We say this partially evaluated code fragment is run in a different *stage* than the rest of that program.

Common programming languages provide a fixed number of stages. For example, C++ provides three stages: preprocessing, template evaluation, and run time. However, in practice these stages do often not suffice. For instance, tools like `lex` or `yacc` [7] generate C/C++ sources, a process which can be regarded as a new, separate stage preceding C preprocessing. Complex code projects even generate source code with the help of scripting languages like Python[1]. This is inelegant for several reasons:

1. This kind of staging requires use of different languages with their own syntax and semantics.
2. The same logical functionality required in different stages must be reimplemented in the syntax of those stages.
3. There is no type safety between stages.

A major step forward in this domain was achieved by introducing MetaML [13]. The language introduced a dedicated construct for staging, allowing the programmer to use an arbitrary number of stages. The code of a deferred stage can be explicitly composed and combined and then invoked through a special `run` command. The presence of staging is, however, statically defined and cannot be easily parametrized.

A diametrically different approach is given by Lightweight Modular Staging (LMS) [10], where staging is not controlled by dedicated language constructs. Instead, it is implicit and automatic, controlled by the type system and function overloading. The source code is usually easy to read and maintain, but it is hard to override the staging rules.

We propose another approach: *Dynamic Staging* which formally is achieved by merely changing the order of beta reduction of otherwise normal programs. This order is explicitly controlled by special **`stage`** variables, which are first-class citizens of the language and can dynamically emerge during evaluation. Terms in the language are not explicitly quoted, but are staged via staging expressions. Most importantly, the programmer can synthesize different code variants suitable for different stages from the same source code fragment.

***Contributions.*** In this paper, we introduce and formally define (Section 3) dynamic staging as a first-class citizen for metaprogramming languages. Dynamic staging does not require quotations and does not influence data types. Novel to our staging approach is that neither the number of stages nor their evaluation order is statically fixed in the program source, but can be specified dynamically. Having staging as a first-class citizen has the following benefits:

- The metaprogramming system is *homogeneous*, that is, the same language is used for all stages. Values and functions can be reused between stages.
- Staging does not require quotation and is independent of data types.

[1] For example, GROMACS' build system makes heavy use of this technique. See `http://www.gromacs.org`.



*GPCE'14*, September 15-16, 2014, Västerås, Sweden
ACM 978-1-4503-3161-6/14/09
http://dx.doi.org/10.1145/2658761.2658774

- All stages share the same lexical environment and variables can be *accessed directly* without any escape syntax.
- Dynamic staging permits *polyvariant specialization* in a straightforward way. The same function can be specialized for different subsets of known parameters.
- Staging may depend on values obtained through arbitrarily complex computation.
- Staging is explicit and the programmer has full control over it.

## 2. Related Work

The common translation process of many programming languages features multiple compilation stages. Thereby, the number of compilation stages is fixed by the programming environment.

For example, C++ distinguishes between preprocessing, template evaluation, and the actual translation of the rest of the C++ program as three distinct stages. In this scenario, each stage even uses a different programming language: the C/C++ preprocessor, the template language and the *core* C++ language. Hence, reusing code between or in the context of different stages requires rewriting of the affected parts of the program.

In contrast to C++, templates in Template Haskell [11] are type safe. It is an extension to Haskell that adds new language features in order to support compile-time meta programming. However, it still requires specific syntax to define and to operate on templates.

One of the first extensions of C in the field of staging is 'C [5]. A programmer can write programs that produce *dynamically generated code*, indicated by a quotation: `'{f(x);}`. The code inside such a quotation is parsed and type-checked at compile-time. The quoted code fragments can be passed as a value, joined together via the `@` operator and specialized through the `$` operator. The approach has several limitations. For instance, quotations cannot be nested and names declared within a quotation cannot be referenced from the outside.

MetaML [13] enhances the concept of 'C in a well-defined and formal way and integrates it into a functional language. A programmer can construct *pieces of code* using `<f x>` which can be defined, combined and executed. Pieces of code can be arbitrarily nested and spliced into surrounding code via the `~` operator. These pieces of code can be executed by a **`run`** command that peels off a single level of angular brackets.

However, the placement of the **`run`** command is limited to ensure that the execution of a piece of code does not get stuck. For instance, the term **`fn`** `x =>` **`run`** `x` is not typable in MetaML and requires either rewriting or an extension of the MetaML type system as described in [14]. Furthermore, it is often difficult to maintain, adapt, and create staged pieces of code in MetaML, as shown in Listing 1.

An approach to enhance MetaML is presented in [8]. It overcomes the limitations of the previously described MetaML via additional operations (e.g. `box` and `unbox`). It tracks so called open and closed code, that is, code with and without free variables. Moreover, it does not suffer from the limitation of the `run` operation anymore.

```
fun back2 f = <fn x => <fn y => ~~(f <x> <<y>>)>>
fun dotF2 n v w =
  if n '>' 0
    then <<(~(lift (nth ~v ~(lift n))) * (nth
         ~~w ~(lift (~lift n)))) + ~~(dotF2 (n-1)
         v w)>>
    else <<0>>;
fun dot n = back2(dotF2 n)
```

**Listing 1:** The staged dot-product function in MetaML [13]. The function can be specialized over the size of the vectors `n`, as well as over the actual value of the second vector.

However, construction and splicing of pieces of code becomes even more complicated since the user needs to insert and maintain even more staging annotations at the right locations.

Several other languages adopted similar staging concept using quotations and annotations. For example, MetaOCaml [2] and Mint [16] are a direct results of bringing the concept of MetaML into OCaml and Java, respectively.

A more pragmatic approach to multiple stages is given by Lightweight Modular Staging (LMS) [10]. It introduces a new type `Rep[T]`. Functions can make use of this type to defer the execution of parts of the code. First the initially staged program is written and then executed afterwards. Instead of executing the intended program code completely, the intended operations are passed to wrappers via custom implementations for different types (e.g. `Rep[`**`int`**`]`). Those wrappers can emit code, create intermediate-representation nodes or perform additional operations. In this approach staging is statically fixed by the type system and specialization is greedy. As a result, additional parameters have to be passed to a function in order to achieve polyvariant specialization. There is no clean mechanism to specify staging rules locally within the body of a single function without affecting its signature.

In the most recent trend, staging is used for domain-specific optimizations. A prominent example is the Terra language [4], which uses a fixed two-stage approach. The first stage uses the untyped Lua language, whereas the second stage uses a typed extension to Lua called Terra. The Lua part can be used to construct fragments of Terra programs, which can be combined and spliced. The resulting Terra code is type checked upon compilation which happens just-in-time during execution of a program.

## 3. Dynamic Staging

In its core, dynamic staging merely defines the order of beta reduction in the program. It is an explicit low-level operation, which, while being verbose, provides more flexibility compared to previous approaches. Dynamic staging can be used to describe quotation-based or type-based staging, but—as we will show—other constructs can be expressed with it in a straightforward way as well; something that is problematic when using previous approaches.

In this section we first intuitively describe our approach to dynamic staging, followed by its formal definition. In Section 4 we provide simple examples, which show how our approach can be used to generate efficient functions, and how different execution plans can be defined depending on the input parameters. Section 5 shows typical use cases for dynamic staging and how the concept can be used to define more practical, higher-level languages. We also show in Section 6 that despite the increased flexibility, the code can be compiled into a highly efficient machine code.

```
float dot(float[] x, float[] y, int n) {
   @n: if (n>0) {
     stage t = *;
     float rest = dot(x,y,n-1);
     @t & x: float xn = x[n];
     @t & y: float yn = y[n];
     @xn & yn & rest:
        float result = rest+xn*yn;
     @t: return result;
   } else
     return 0.0;
}
```

**Listing 2:** The same staged dot-product function in Stage-C which we introduce in this paper. The function can be specialized over the size of the vectors `n`, as well as over the actual value of *either* of the vectors.

$$
\begin{array}{lr}
v ::= & \text{(value)} \\
\quad c \in \{0, 1, 2, \ldots, \texttt{true}, \texttt{false}, \ldots\} & \text{(constant)} \\
\quad \lambda\overline{x}.b & \text{(function)} \\
\quad x & \text{(parameter)} \\
b ::= & \text{(body)} \\
\quad v\ \overline{v} & \text{(application)} \\
\quad \texttt{fix}\ x = \lambda\overline{x}.b\ \texttt{in}\ b & \text{(fix)}
\end{array}
$$

**Figure 1:** Syntax of CPS-based lambda calculus.

### 3.1 Overview

We define the dynamic staging concept as an extension of a Continuation Passing Style (CPS) functional language. We have chosen CPS because:

- The canonical order of execution in a CPS program is well defined.
- In standard CPS [1, 12] every expression is contained within its own unique lambda function, thus becoming its complete *body*. There are no subexpressions. This correspondence expression↔lambda is important for our definition of staging.

The programmer can annotate function bodies with boolean *staging expressions*. When the staging expression evaluates to $\top$, we denote the annotated body as *active*. In each evaluation step we find the innermost active body and perform a beta reduction on that body.

The staging expression is a boolean expression over an arbitrary set of parameters. When evaluating it, however, we do not check the values of the parameters, but only if the parameters are concrete values or remain unbound. Known values (constants and lambdas) are $\top$, while unbound values are $\bot$.

Every lambda function defines a special *implicit staging parameter* associated with the function. When the function is invoked, this implicit staging parameter becomes set. A staging parameter does not hold any meaningful value. The mere fact that it is being set is what matters, as it impacts the result of staging expressions where the parameter is used.

In addition to the implicit staging parameter, functions can accept an arbitrary number of explicit staging parameters. Implicit and explicit staging parameters can be freely passed as arguments to subsequent functions.

### 3.2 CPS-based Lambda Calculus

#### 3.2.1 Syntax

The syntax of the *CPS-based lambda calculus* (Figure 1) is similar to the untyped lambda calculus [9, Chap. 5]. However, there exist two major differences:

1. Functions do not return in CPS, thus they never return a value. A function body is a single application or a *fix construct*. The whole remainder of the program is contained either within the body, or is passed as an argument, called a *continuation*.
2. For the same reason, subexpressions cannot be expressed and we cannot curry functions. Hence, we allow arbitrarily long parameter lists.

We further define a fixed-point combinator directly in the language to support simple recursion: $\texttt{fix}\ x_f = \lambda\overline{x}.b\ \texttt{in}\ b_f$ allows recursive use of $x_f$ in $b$.

$$
\begin{array}{lr}
v ::= & \text{(value)} \\
\quad c \in \{0, 1, 2, \ldots, \texttt{true}, \texttt{false}, \ldots\} & \text{(constant)} \\
\quad \lambda\,\texttt{[}y\texttt{]}\,\overline{x}.b & \text{(function)} \\
\quad x & \text{(parameter)} \\
\quad y & \text{(staging parameter)} \\
e ::= & \text{(staging expression)} \\
\quad \top & \text{(true)} \\
\quad \bot & \text{(false)} \\
\quad v & \text{(value)} \\
\quad e\ \texttt{and}\ e & \text{(and expression)} \\
\quad e\ \texttt{or}\ e & \text{(or expression)} \\
\quad \texttt{not}\ e & \text{(not expression)} \\
b ::= & \text{(body)} \\
\quad \texttt{[}e\texttt{]}\,v\ \overline{v} & \text{(application)} \\
\quad \texttt{[}e\texttt{]}\,\texttt{fix}\ x = \lambda\,\texttt{[}y\texttt{]}\,\overline{x}.b\ \texttt{in}\ b & \text{(fix)}
\end{array}
$$

**Figure 2:** Syntax of staged CPS-based lambda calculus.

Finally, we directly include constants in order to make the language more practical. We use the common notation $\overline{a}$ to denote a list $a_1, \ldots, a_n$.

#### 3.2.2 Semantics

The following rules, similarly to Pierce [9, Chap. 5.3], describe the operational semantics of the language:

$$
\frac{}{(\lambda\overline{x}.b)\ \overline{v} \rightarrow [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\, b}
$$

$$
\frac{}{\texttt{fix}\ x_f = \lambda\overline{x}.b\ \texttt{in}\ b_f \rightarrow [x_f \mapsto \lambda\overline{x}.\texttt{fix}\ x_f = \lambda\overline{x}.b\ \texttt{in}\ b]\, b_f}
$$

We require all names in the program to be unique in order to circumvent name capture issues in the rules. Apart from the differences to untyped lambda calculus mentioned earlier, the rules are standard.

### 3.3 Staging in CPS-based Lambda Calculus

In traditional CPS in each step the beta reduction is applied on the top-most body only. This causes the program to be evaluated in the order from top to bottom. A different approach may permit a beta reduction step to occur indeterminately on any body within the program, which causes a reordering of the aforementioned beta reduction. With the concept of dynamic staging we let the programmer explicitly specify the order of the applied reduction steps.

#### 3.3.1 Syntax

In order to extend the presented calculus with staging we introduce a new syntactic category *staging expressions* (Figure 2). These are the usual boolean expressions and all applications and fix constructs are annotated with them. Staging expressions may reference parameters or *staging parameters*.

#### 3.3.2 Semantics

Before diving into the whole semantics of staged CPS-based lambda calculus, we need to define some supporting relations. These relations find the deepest bodies that are ready to be executed and mark them as active, ensuring that the deepest bodies are executed first as described in Section 3.1.

*Active.* The following judgment determines whether a staging expression is *active* ($A$), that is, the staging expression evaluates to $\top$:

$$\frac{}{A(\top)} \qquad \frac{}{A(c)} \qquad \frac{}{A(\lambda\,[y]\,\overline{x}.b)}$$

$$\frac{A(e_1) \quad A(e_2)}{A(e_1 \texttt{ and } e_2)} \qquad \frac{A(e_1) \vee A(e_2)}{A(e_1 \texttt{ or } e_2)} \qquad \frac{\neg A(e)}{A(\texttt{not } e)}$$

Note that there do not exist rules for $\bot$, $x$, and $y$. This means, a term $x$ is considered inactive as long as it is unresolved. The substitution semantics given below replaces all occurrences of $x$ as soon as the parameter is bound, rendering the emerging staging subexpression active. Similarly, any staging term $y$ is considered inactive. As soon as the corresponding function is applied, all occurrences of $y$ are replaced with $\top$ rendering the emerging staging subexpression active, too. Informally, we say a body is active, when its staging expression is active.

*Waiting.* A term (a value or a body) is called *waiting* ($W_v$, $W_b$) when it does not contain any arbitrarily nested active staging expression.

$$\frac{}{W_v(c)} \qquad \frac{}{W_v(x)} \qquad \frac{}{W_v(y)} \qquad \frac{W_b(b)}{W_v(\lambda\,[y]\,\overline{x}.b)}$$

$$\frac{\neg A(e) \quad W_v(v) \quad \overline{W_v(v)}}{W_b(\,[e]\,v\,\overline{v})} \qquad \frac{\neg A(e) \quad W_b(b) \quad W_b(b_f)}{W_b(\,[e]\,\texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b\,\texttt{in}\,b_f)}$$

*Rebuilding.* A term is only allowed to execute, when it is active and all its subterms are waiting. All other terms not being executed produce a new term by recursively decomposing all subterms and reassembling them again with the newly produced subterms:

$$\frac{}{c \to_v c} \qquad \frac{}{x \to_v x} \qquad \frac{b \to_b b'}{\lambda\,[y]\,\overline{x}.b \to_v \lambda\,[y]\,\overline{x}.b'}$$

$$\frac{\neg(A(e) \wedge W_v(v) \wedge \overline{W_v(v)}) \quad v \to_v v' \quad \overline{v \to_v v'}}{[e]\,v\,\overline{v} \to_b [e]\,v'\,\overline{v'}}$$

$$\frac{A(e) \quad W_v(v) \quad \overline{W_v(v)} \quad [e]\,v\,\overline{v} \to b'}{[e]\,v\,\overline{v} \to_b b'}$$

$$\frac{\neg(A(e) \wedge W_b(b) \wedge W_b(b_f)) \quad b \to_b b' \quad b_f \to_b b'_f}{[e]\,\texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b\,\texttt{in}\,b_f \to_b \to_b [e]\,\texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b'\,\texttt{in}\,b'_f}$$

$$\frac{A(e) \quad W_b(b) \quad W_b(b_f) \quad [e]\,\texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b\,\texttt{in}\,b_f \to b'}{[e]\,\texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b\,\texttt{in}\,b_f \to_b b'}$$

*Evaluation.* We enhance the original evaluation rules by including staging expressions. In particular, we replace all occurrences of $y$ with $\top$ in the case of an application:

$$\frac{}{[e]\ (\lambda\,[y]\,\overline{x}.b)\,\overline{v} \to [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, y \mapsto \top]\,b}$$

$$\frac{}{[e]\ \texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b\,\texttt{in}\,b_f \to \to [x_f \mapsto \lambda\,[y]\,\overline{x}.[y \texttt{ and } e]\ \texttt{fix}\,x_f = \lambda\,[y]\,\overline{x}.b\,\texttt{in}\,b]\,b_f}$$

*Staged Execution.* Finally, we keep rebuilding the program as long as we find active terms. The program may terminate if it collapses to an application of a built-in *exit* function called with argument $v$. Then, $v$ is the result of the program:

$$\frac{b \to_b b'}{\lambda.b \to_p \lambda.b'} \qquad \frac{b \to_b \mathit{exit}\ v}{\lambda.b \to_p v}$$

### 3.4 Typing

Reading the previous paragraph from a different point of view, reveals that program execution may get stuck if there exist no more active subterms. So far, our considerations were untyped. There arises the question whether we could design a sound type system for the language, that is, we could statically guarantee that well-typed programs never get stuck.

However, one of the key features of the presented staging approach is that whether a stage becomes active or not depends on staging parameters ($y$). Their value may depend on the execution of an arbitrary function $f$. Since we impose no restriction on the contents of $f$ and the language is Turing-complete and due to the halting problem [15] it is in general undecidable to statically foresee the result of $f$ and the value of $y$. Hence, it is not possible in general to design a sound type system without significantly cutting the power of the calculus (see also Section 6.2.3).

### 3.5 Observations

*Staging List.* A stage parameter can be seen as a *staging list* containing references to all bodies that need to be checked for possibly becoming active when the stage parameter itself is set. When stage argument $a$ is explicitly passed to a function as parameter $q$, the bodies in the list $q$ become added to the list $a$. In this view staging lists are passed in the reverse direction: from the callee to the caller.

*Natural Staging.* It is a common case for a given function $\lambda$ with an implicit staging parameter $s$ to have its body annotated by $[s]$:

$$\lambda\,[s]\,\overline{x}.\,[s]\,v\overline{v}$$

Assuming that this is the only use of $s$, when the lambda is invoked, its body is the only new body that becomes active. Consequently, the body is evaluated in the next execution step similarly to a normal CPS program.

*Deepest First.* When multiple bodies are active, the deepest one is always executed first. This is the case because the *Evaluation* rules are applied only for active bodies whose subterms are waiting, containing no further active bodies.

This ensures that any body that is made active will never be duplicated due to parameter-argument substitution during beta reduction of some parent body. Consequently, once a body is made active, it is executed exactly once.

## 4. A Language for Staged CPS

### 4.1 Syntax

In this section we introduce a functional language called *DeepCPS*, suited for CPS programming with staging annotations. The syntax is similar to a normal functional language, but with noticeable changes to facilitate more compact and readable code, as summarized in Figure 3:

- The lambda function specifies its implicit staging parameter in square brackets.
- While functions are of arbitrary arity, the invocation does not need any parenthesis. This is because subexpressions are not allowed in CPS, thus currying cannot occur.
- Since lambda functions can be member of a multi-argument list, their body is put into curly brackets. The braces can be dropped only when a function is the last argument in the list.
- Application is preceded by the staging expression, put in between `@` and `:`. Lack of such annotation implies natural staging.
- Sections of code that do not require staging or CPS are going to be written in pseudocode for the sake of readability. These sections are highlighted in blue and the following lambda function is invoked with the resulting value.

Basic syntax:

| | | |
|---|---|---|
| $\lambda[y]\overline{x}.b$ | `(x1, x2, ...)[y]{b}` | (function) |
| $x$ | *type name* | (parameter) |
| $\top\ \bot$ | `always never` | (staging values: true, false) |
| `and or not` | `& \| !` | (staging boolean operators) |
| $[e]\,v\,\overline{v}$ | `@e:v v1 v2 ...` | (application) |
| $[e]$`fix` $x = f$ `in` $b$ | **`fix`** `def x` *function b* | (fix) |

Syntactic sugar:

| | | |
|---|---|---|
| $\lambda[y]\overline{x}.[y]v\overline{v}$ | `(x1, x2, ...){ v v1 v2 ... }` | (natural staging) |
| $[y_1]\ \lambda[y_2]x.b\,v$ | `@y1:`**`let`** `[y2] x v b` | (let construct) |
| $\lambda[y]\overline{x}.b$ | `(x1, x2, ...)[y]`$b$ | (function as last argument) |
| $p\,\lambda[y]x.b$ | *p* `(x) b` | (non-CPS expression $p$) |

**Figure 3:** Syntax of lambda calculus and DeepCPS.

```
let power (float base, int exp, fn[float] cont) {
  exp==0 (bool b)
  if b () {
    cont 1
  } ()
  exp mod 2 == 1 (bool b)
  if b () {
    exp-1 (em)
    power base em (part)
    part*base (result)
    cont result
  } () {
    exp/2 (eh)
    power base eh (part)
    part*part (result)
    cont result
  }
}
let power72 (float base, fn[float] cont) {
  power base 72 (result)
  cont result
}
```

**Listing 3:** An integer power function and a special case where the input `base` value is raised to the power of 72.

In DeepCPS, values are expressed as numeric constants, names of previously declared parameters, or one of the intrinsic functions. A conditional `if` is also an intrinsic function, rather than a fixed specialized language construct: `if cond true_branch false_branch` accepts a normal (not staging) boolean value and invokes either `true_branch` or `false_branch`.

We use basic numeric types (**`bool`**, **`int`**, **`float`**) as well as aggregate type `[...]` and function type **`fn`**`[...]`. When the type can be easily deduced from the context or is irrelevant for the given example, we choose to omit it. In addition, each application is started a new line for readability.

### 4.2 Static Use of Staging

In the following, we use the integer power function of Listing 3 as a simple running example. `power72` is a specialized function for a single known exponent. However, it does not benefit from this knowledge and it still invokes the same generic `power` function every time `power72` is invoked.

We assume for now that a `powergen` function is provided, that generates specialized code when the exponent is provided while the `base` value is unknown. The `powergen` function should be executed once in the context of a function `power72`, before the latter is invoked. We are not changing the logic of those functions, but merely the order of their execution. This can be achieved through static usage of staging variables as shown in Listing 4 and graphically visualized in Figure 4.

```
let [def] powergen (...) { ... }
@def: let power72 (float base, fn[float]
    cont)[call] {
  @def:  powergen base 72 (result)[]
  @call: cont result
}
```

**Listing 4:** Invoking `powergen` before `power72` is invoked. As soon as `powergen` is defined by the **`let`** instruction, the staging variable `def` becomes active, which triggers all instructions marked by `@def`, starting from the deepest one. Only once the `powergen` finishes, the modified anonymous function is bound to the name `power72`.

At the start of the program, the first **`let`** instruction is invoked, which defines the `powergen` function. Moreover, this invocation activates the `def` staging variable. Two instructions, as indicated by red arrows, become active: the direct follow-up that binds a function definition to the name `power72` and the call to `powergen` within the body of that function.

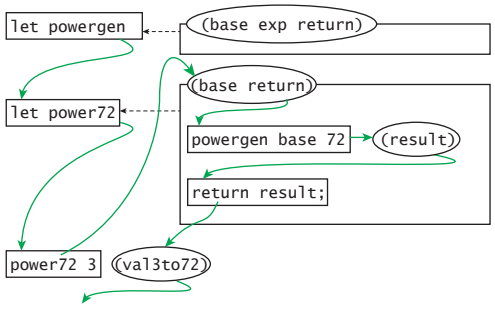
The `powergen` call has a deeper nesting level and is, hence, executed first. At the time of the call the parameter `base` is not yet known and `powergen` is partially evaluated, producing code that is spliced into the definition of `power72`. The `powergen` at the end of its execution invokes its continuation function `(result)[]`. The empty bracket indicates that the implicit staging variable is not used anywhere and no further instruction is scheduled for execution as a result of this. At this point, the previously annotated `@def:`**`let`** instruction can be executed, which binds the partially evaluated, anonymous function to the name `power72`.

Note that the spliced code of the partially evaluated `powergen` function should be executed when `power72` is invoked. For that reason, when `[call]` gets active, the execution should not jump directly to the `cont result` instruction. It is indeed an error, introduced for the sake of simplicity of the example. We are now going to show how to correct this error.

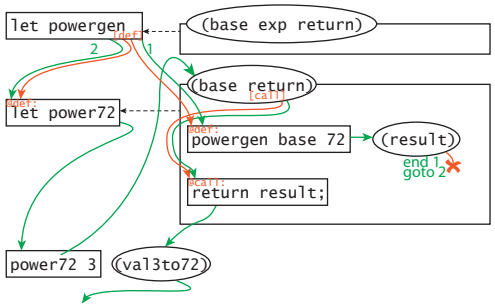
### 4.3 Staging as a Parameter

We now focus on `powergen`, which is shown in Listing 5. The computational logic of the function does not differ from `power`, but care has to be taken on which values are known and which are not:

- Although not indicated by the type system, an argument with unknown value can be passed as `base`. We name these kind of parameters as *meta parameters*.




**(a)** unstaged CPS




**(b)** staged CPS

**Figure 4:** Graphical representation of the instruction schedule using (a) unstaged and (b) staged CPS. Green arrows show the order of execution, red mark the staging.

- However, some instructions can be executed only when both `exp` and `base` values are known. For that reason, we introduce the `full` staging parameter, which will be active if both values are known.
- When `powergen` is partially evaluated, it calls `cont` with the parameter `result` which will be bound later. As soon as `result` becomes known (bound), the associated **`stage`** parameter `recf` becomes active.

This passing of staging parameters allows the programmer to specify which portions of the code can execute immediately, and which can execute only once another function was called. Thus, setting variables to concrete values is performed upon the call.

In the example of Listing 5, the code that depends only on `exp`—the conditions and recursion—is evaluated immediately. The recursive call passes the value `part` and the stage value `recf` to its continuation. The stage value indicates when `part` will evaluate to a concrete value. The following multiplication instruction is being postponed and the `cont` is invoked with a yet unknown value `result`.

The specialized `power72` function needs to be adjusted to use additional staging parameters. The additional **`stage`** parameter `[call]`, which becomes active when `power72` is invoked and the concrete value of `base` is known, is passed to the `powergen` function. Similarly, the continuation of `powergen` has a parameter `recf` which becomes set when `result` is a concrete value. This stage is used in `power72` as `recf` to stage the final `cont` statement.

### 4.4 Staging on Variables

It is very common, as in our example, to accompany a meta parameter with a staging parameter indicating when the meta parameter will hold a concrete value. To facilitate this common case we can use meta variables within the staging annotation directly.

```
let [def] powergen (float base, int exp, stage
    full, fn[float, stage] cont) {
  exp == 0 (bool b)
  if b () {
    cont 1 always
  } ()
  exp mod 2 == 1 (bool b)
  if b () {
    exp-1 (em)
    powergen base em full (part, recf)[rec]
    @full & recf: part * base (result)[res]
    @rec: cont result res
  } () {
    exp/2 (eh)
    powergen base eh (part, recf)[rec]
    @recf: part * part (result)[res]
    @rec: cont result res
  }
}
@def: let power72 (float base, fn[float]
    cont)[call] {
  @def:  powergen base 72 call (result, recf)[]
  @recf: cont result
}
```

**Listing 5:** Explicitly staged version of the `powergen` function. `base` can accept an unbound name which is resolved to a concrete value no later than at a stage `full`. The multiplication operations are annotated by staging expressions, ensuring that the operation is done only when the parameters are concrete values. The recursive call to `cont`, however, is allowed to be invoked early. This allows resolving the recursion while working on the symbolic values.

This follows the semantic rules of *Active* as defined in Section 3.2.2: a parameter with a bound concrete value becomes equivalent to $\top$, while a unknown value corresponds to $\bot$. This allows to simplify the previous example by removing the explicit stage parameter from `powergen`, as shown in Listing 6.

```
let powergen (float base, int exp, fn[float]
    cont) {
  exp == 0 (bool b)
  if b () {
    cont 1
  } ()
  exp mod 2 == 1 (bool b)
  if b () {
    exp-1 (em)
    power base em (part)[rec]
    @base & part: part * base (result)
    @rec: cont result
  } () {
    exp/2 (eh)
    power base eh (part)[rec]
    @part: part * part (result)
    @rec: cont result
  }
}
@power: let power72 (float base, fn[float] cont)
    {
  @power:  powergen base 72 (result)
  @result: cont result
}
```

**Listing 6:** `powergen` function with staging on parameters, which allows to remove almost all explicit usage of **`stage`** parameters. `@part`, `@base`, `@power`, and `@result` are read "as soon as given parameter is set to a concrete value".

At this point, the signature of `powergen` does not differ from `power` and can be used in all contexts where the original `power` function was used.

## 5. Applications of Dynamic Staging

Having all the concepts of dynamic staging explained, we focus now on practical applications. We show typical patterns of staging in DeepCPS and additionally show how they can be mapped to *Staged-C*—a higher-level C-like language with staging, so that the flexibility is maintained but the user would not have to deal with low-level CPS explicitly.

### 5.1 Staged-C

In Staged-C we use the same syntax as in DeepCPS to annotate instructions that need to be executed at a different stage. The annotation resembles the goto label. Instructions without an annotation are executed immediately after the preceding one.

In Staged-C the implicit stage parameter associated with a function is named the same as the function. We do not use the `[name]` syntax after the parenthesis.

A special stage `*` indicates the stage of the preceding instruction in the program text.

When multiple instructions are staged upon the same variable in Staged-C, the *earlier* one is executed first—unlike in DeepCPS where we required the deepestmost action to be taken first. This change can be easily achieved in DeepCPS by staging the next instruction upon the previous one:

DeepCPS:

```
@s:  f1 ()[s1]
     ...
@s1: f2 ()[s2]
     ...
@s2: f3 ()[s3]
     ...
```

Stage-C:

```
@s: f1();
    ...
@s: f2();
    ...
@s: f3();
    ...
```

The DeepCPS order was necessary at the low-level to handle higher-order functions properly, but Staged-C does not support higher-order functions.

### 5.2 Function Specialization

Unlike in MetaML, the process of function specialization does not differ from a normal function call. A function does not return an object of type *code*, which would then be invoked with `run`. Instead, upon invocation the function is inlined and the deferred code remains as a "left over" from the call, in the context of the caller.

In order to specialize a function, one encapsulates the call to the generic version in another function. The generic version is then invoked before the anonymous lambda function is bound to a name. As a result, a specialized version is inlined into the body of the lambda function and does not differ from a version that would be hand-written at that location in the code.

Because this is a very common pattern, Staged-C introduces a new syntax for it. We permit a function to be defined as a call to a generic version using angular brackets:

DeepCPS:

```
let generic(params) ...
let special(prms,ret) {
  @generic:
    generic args (res)
  @res: ret res
}
```

Stage-C:

```
ret_type
    generic(params);
ret_type special(prms)
    = generic<args>;
```

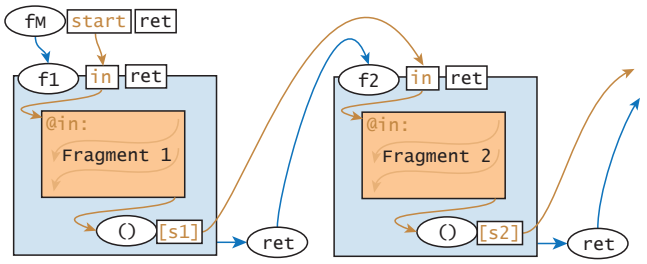
The arguments within the brackets may include any previously declared variable in the current scope, as well as the parameters of the specialized function. The angular-bracket call is *not* considered to be an invocation. The generic function is merely inlined and only code staged upon known variables is executed.

```
let f1(stage in, fn[stage] cont) {
  ... normal code ... [jmp]
  @in: ... fragment 1 code ... [s1]
  @jmp: cont s1
}
let f2(stage in, fn[stage] cont) {
  ... normal code ... [jmp]
  @in: ... fragment 2 code ... [s2]
  @jmp: cont s2
}
let f3 ...
let f4 ...
let fM(stage start, fn[stage] ret) {
  f1 start (s1)
  f2 s1 (s2)
  f3 s2 (s3)
  f4 s3 (s4)
  ret s4
}
```

**(a)** Unconnected code fragments.

```
@start: ... fragment 1 code ... [s1]
@s1: ... fragment 2 code ... [s2]
@s2: ... fragment 3 code ... [s3]
@s3: ... fragment 4 code ... [s4]
...
```

**(b)** Generated code after fragment chaining.




**(c)** Graphical representation.

**Listing 7:** Fragment chaining pattern:
**(a)** Function `f1` and `f2` contain a code fragment that is staged upon the **`stage`** parameter `in` and the stage of the fragment's last instruction is returned. These functions are invoked in a sequence, chaining the code fragments by passing the result of one function as an input for the next one.
**(b)** The result of the invocation of the master function `fM`.
**(c)** Graphical representation of the fragment chaining pattern. Blue blocks represent the code being executed immediately, orange blocks are the deferred fragments chained together.

### 5.3 Fragment Chaining

One of the capabilities of CPS and dynamic staging is to connect two initially unrelated blocks of code, so that in the next stage they are executed one after another. This is achieved through the *fragment chaining* pattern:

Consider a set of functions `f1`, `f2`, ..., `fn`, each of type **`fn[stage, fn[stage]]`**. Each function contains a code fragment staged with respect to its **`stage`** parameter, creating the *staged fragment* pattern as described before. Therefore, we call such functions *container functions*. These functions can be chained together by using the return value from a function as input for a subsequent function.

For example, the master function `fM` in Listing 7 (a) invokes functions `f1`, ..., `fn` in that order. When `fM` is invoked, all the `fi` are executed whereby the staging parameters get resolved. As a

```
let sfor(stage pstage, int from, int to,
    fn[stage, int, fn[stage]] body, fn[stage]
    end) {
  from<to (bool continue)
  if continue () {
    body pstage from (stage nstage)
    from+1 (next)
    sfor nstage next to body end
  } () {
    end pstage
  }
}
```

**Listing 8:** CPS-style `for` loop with its body invoked at a separate stage. This allows the loop to be explicitly unrolled.

result, all code fragments in $fi$ become connected through staging as shown in Listing 7 (b).

### 5.4 Staged For Loop

As a special case of fragment chaining consider `sfor` in Listing 8. The function implements a C-like `for` loop in CPS style, iterating from integer value `from` to `to`, invoking its body with each iteration value as an argument. Unlike a typical loop, however, it additionally passes a **`stage`** value to the `body`, and receives a new **`stage`** value from the `body` through the continuation call.

The function `sfor` acts as master function. It iteratively invokes its container function `body`. The author of `body` stages a part of its content on the input stage parameter, forming the fragment block.

Upon the execution of `sfor`, the loop semantics are resolved while the deferred fragment blocks from `body` remain as a code fragment, staging one upon another in a chain. As a result we obtain an unrolled version of the loop.

Naturally, in Staged-C, `for` loop is given as a keyword rather than a function. Arbitrary number of parameters, including staging parameters can be passed into the body of the loop. By having the for loop at a different stage than a fragment of its body the programmer can explicitly trigger unrolling of the loop.

### 5.5 Staged Convolution

A more practical example of using the staged for loop are convolutions as used in image processing. As an example, we consider the 1D convolution in Listing 9 (a): The generic `convolve` function accepts three pointers to memory: the input image `in`, the kernel data `K`, and the output image `out`. When dealing with memory, automated tools have to be very conservative when doing optimizations. With dynamic staging, however, the programmer can explicitly define which kind of specialization should be applied.

As a result of partial evaluation, we obtain a convolution with an unrolled inner loop (Listing 9 (c)), although the number of iterations is unknown in the original code. Note that address calculation for indexing `K` and associated loads have been completely removed. Applying constant folding afterwards would remove all multiplications, in turn eliminating the unnecessary memory access.

### 5.6 Fast Fourier Transform

Another example in which staging is useful is the Fast Fourier Transform. We have implemented the recursive Cooley-Tukey algorithm [3] (Listing 10), with the assumption that the input contains $2^k$ elements. All functions that are recursively called are specialized for a concrete value of `size`.

We introduce a local stage variable `unroll` which gets associated either with `sh` or `image` depending on the value of `sh`. Its value controls if the recursive calls and the for loop are unrolled early, during the specialization, or are left intact to be resolved only

```
void convolve(float* in, int size, float* K, int
    range, float* out) {
  for (int x=0; x<size; ++x) {
    float v = 0;
    @K & range:
      for (int j=-range; j<=range; ++j) {
        float kv = K[range+j];
        @* & in:
          v += in[x+j]*kv;
      }
    out[x] = v;
  }
}
```

**(a)** Staged 1D convolution.

```
void conv_spec(float* in, int size, float* out) =
  convolve<in, size, {-1.0, 0.0, 1.0}, 1, out>;
```

**(b)** Specialization for $K = [-1\ 0\ 1]$.

```
void conv_spec(float* in, int size, float* out) {
  for (int x=0; x<size; ++x) {
    float v = 0;
    v += in[x-1]*-1.0;
    v += in[x+0]*0.0;
    v += in[x+1]*1.0;
    out[x] = v;
  }
}
```

**(c)** Specialized code for $K = [-1\ 0\ 1]$.

**Listing 9: (a)** staged 1D convolution, which can be specialized for a given kernel `K`. **(b)** specialization of the convolution. **(c)** specialized code for $K = [-1\ 0\ 1]$.

when `image` is provided. This kind of computation on stages is possible because staging is a first-class citizen of the language.

If specialization works automatically and greedily like in LMS, trying to use all the information available, both the recursion and loop will be unrolled for all values of `size`, which will lead to code having $O(n \log n)$ instructions. We use staging annotations to prevent that and precisely specify how much of the code we want to be specialized.

## 6. Evaluation and Discussion

In this section, we evaluate our staging concept in terms of performance and discuss other aspects like expressiveness and program correctness.

### 6.1 Evaluation

In order to evaluate our staging concept with respect to performance, we have implemented a CPS-based language interpreter and compiler. We can interpret programs using the staging rules as described before. A intrinsic function `compile` is provided that takes a (partially-evaluated) function as an argument and compiles using LLVM [6].

We have selected a few algorithms and implemented them using both our language and C++:

1. Power function computing $x^{72}$ of Listing 6.
2. A convolution of an unknown 1D data set with a kernel of $[-1, -2, 0, +2, +1]$.
3. The same convolution, but with additional boundary handling (mirror).
4. Fast Fourier Transform (FFT) of a 1D signal of $2^{24}$ elements.

**Table 1:** Absolute execution times of different implementations of the power function and 1D convolution. Two variants of the convolution are considered, one using mirroring as boundary handling (BH) and one without boundary handling.

| | power | convolution BH: none | convolution BH: mirror | FFT |
|---|---|---|---|---|
| Number of iterations | $10^8$ | $10^6$ | $10^6$ | 1 |
| Standard C++ | 1402 ms | 1079 ms | 1171 ms | 2510 ms |
| Templated C++ | 192 ms | 419 ms | 421 ms | 551 ms |
| CPS without staging | 1685 ms | 1012 ms | 1047 ms | 2584 ms |
| CPS with staging | 194 ms | 413 ms | 420 ms | 573 ms |

```
void reindex(float* image, int size) { ... }
void sfft(float* image, int size) {
  @size:
    if (size == 1) return;
    int sh = size/2;
    stage unroll = sh<=4 ? sh : image;
    void sffth(float* img) = sfft<img, sh>;
  @* & unroll:
    sffth(image);
    sffth(image + sh);
  @sh:
    float wtemp = sin(pi/sh);
    float wpr = -2.0 * sqr(wtemp);
    float wpi = -sin(2*pi/sh);
    float wr = 1.0;
    float wi = 0.0;
  @* & unroll:
    for (int i=0; i<sh; i+=2) {
      @* & image:
        tempr = image[i+sh]*wr -
            image[i+sh+1]*wi;
        tempi = image[i+sh]*wi +
            image[i+sh+1]*wr;
        image[i+sh] = image[i]-tempr;
        image[i+sh+1] = image[i+1]-tempi;
        image[i] += tempr;
        image[i+1] += tempi;
      @i:
        wtemp = wr;
        wr += wr*wpr - wi*wpi;
        wi += wi*wpr + wtemp*wpi;
    }
}
void fft(float* image, int size) {
  reindex(image, size);
  @size:
    sfft(image, size);
}
```

**Listing 10:** The implementation of the recursive Cooley-Tukey algorithm for the Fast Fourier Transform. When `size` is known, all recursive functions can be partially evaluated, eliminating the extensive trigonometric functions. For big values of `size` the recursion itself and the loop are left intact however, to keep the produced code short.

These algorithms are implemented in the following 4 ways:

1. Standard C++ dynamic code.
2. C++ with metaprogramming, partially executed at compile-time.
3. In our CPS language without staging, producing dynamic code.
4. In our CPS language with staging, partially executed at compile-time.

For boundary handling, we specialize the convolution on different regions of the data set: instead of checking the boundary condition for every data point, we peel off the iterations that require boundary handling and specialize these iterations. This results in three loops iterating over the left-most, inner, and right-most part of the data set. Hence, the inner loop does not require checks for boundary handling. The corresponding implementation using C++ templates requires the body of the convolution to be available as a lambda function (introduced with C++11), which is passed to templates that peel off the loop iterations.

The naive implementation of the Fast Fourier Transform uses a recursion while the templated/staged version unwinds it at compile time.

C++ sources have been compiled using clang 3.3 into LLVM bitcode [6] (`clang++`). The staged implementations are compiled into LLVM bitcode using our compiler. The LLVM bitcode is then compiled and linked into native code using LLVM with optimizations performed on the bitcode (`opt -O3`).

All the produced functions have been called from the same main loop, repeating the call several times. The parameters were flagged as `volatile` to prevent any optimization between loop iterations. The produced executables have been run on a computer equipped with an Intel i7-2600K 3.4 GHz CPU and 8 GB of DDR3 (1333 MHz) memory, running 64-bit Ubuntu 12.04.2 LTS. Each executable was run 5 times and the average timing was used.

From Table 1, the following can be seen:

- Partially evaluated code can perform significantly better than unspecialized code.
- Using our LLVM back end to compile our implementations using staging produces code that is comparable in performance to the existing and well-adopted C++ compiler.

Using dynamic staging, the programmer simply had to add staging annotations to generic implementations. There was no need to use multiple languages like in Terra, or to write different code for specialization like in C++ metaprogramming.

### 6.2 Discussion

#### 6.2.1 Staging Expressiveness

Using our staging approach, we can express quoted code blocks, known from MetaML and other staging languages. This can be achieved by a pattern similar to fragment chaining in Listing 7. Dynamic staging, however, steps beyond selecting instruction blocks for earlier/later execution. It allows repeated execution of the same function in the context of different stages. The author of the function can write generic code with respect to staging, and the caller can determine its execution plan using stage parameters.

#### 6.2.2 Cross-stage Persistence

As staging is defined merely as a change of order of the beta reduction steps, there is no distinct transition from one stage to

another. Values created in one stage remain live as long as the code referencing them can potentially be executed. There is no special action required from the user to achieve that.

#### 6.2.3 Program Correctness Using Staging

It is the programmer's responsibility to ensure program correctness under staging. Careless staging may cause programs to return incorrect values, to never terminate, or to produce infinite intermediate code. In particular, the programmer has to pay attention when staging over *branch* nodes, since it may lead to code execution in a branch that is ultimately not taken. These problems cannot be circumvented since staging permits the programmer to specify what *he* considers to be safe, even if the compiler is not able to prove the correctness of the desired transformation.

Incorrect staging can also lead to an error, when either no instructions are active, or an active instruction requires a concrete value when only an unbound parameter is provided. As discussed in Section 3.4, a type system cannot in general detect these cases. Consider the following example of a recursive function `fun`:

```
fix fun(int m, int n, int p, fn[] return) {
  (cont : fn[stage]) {
    ackermann(m,n)>125 (bool b)
    if b () { cont always }
         () { cont p }
  } (stage s)
  @s: p-1 (q)
  fun m p q return
} ...
```

The stage parameter `s` depends on the result of a computation of the non-primitive-recursive Ackermann function. In general a compiler cannot determine whether this invocation will terminate or not and cannot predict whether stage `s` will become active or not. Consequently, all instructions staged upon `s` are not verifiable through a static analysis.

## 7. Conclusions and Future Work

In this paper, we introduced a novel staging approach called *dynamic staging* that can be integrated into other languages. Our solution introduces staging as a first-class citizen. This allows the programmer to flexibly control the process of partial evaluation. The same function can be reused in the context of different stages and specialized in different variants. Staging may also depend on result of some other computation.

We provide a formalization of our approach including full semantics. We are able to express other staging concepts, like the general concept of quotation, with our approach. We hope that our formalization can be used in the future as a basis for defining and reasoning about all forms of staging.

In our experiments we have shown that our compilation pipeline employing the dynamic staging concept can compete with existing compilers in terms of produced code efficiency, while starting from a completely generic code base.

***Future Work.*** Languages supporting staging may be of great use when creating Domain Specific Languages (DSLs). This is because DSLs often perform domain-specific semantic analysis and optimizations. These processes as well as code generation can be delegated to different stages. We believe that our approach may be particularly useful in this context because of the homogeneous syntax and code reusability between stages. In particular, the fragment chaining pattern (see Section 5.3) allows to create building blocks for DSLs. Moreover, having staging as a first-class construct should permit DSLs to also support their own kind of staging—explicitly or implicitly.

## Acknowledgments

This work is partly supported by the Federal Ministry of Education and Research (BMBF), as part of the ECOUSS project, and by the Intel Visual Computing Institute (IVCI) Saarbrücken. It is also co-funded by the European Union (EU), as part of the Dreamspace project.

## References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[2] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing Multi-stage Languages using ASTs, Gensym, and Reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, pages 57–76. Springer, Sept. 2003.

[3] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 105–116. ACM, June 2013.

[5] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 131–144. ACM, 1996.

[6] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, Mar. 2004.

[7] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[8] E. Moggi, W. Taha, Z.-E.-A. Benaissa, and T. Sheard. An Idealized MetaML: Simpler, and More Expressive. Lecture Notes in Computer Science, pages 193–207. Springer, 1999.

[9] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1.

[10] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 127–136. ACM, Oct. 2010.

[11] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM, 2002.

[12] G. Sussman and G. Steele. *Scheme: An Interpreter for Extended Lambda Calculus*. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.

[13] W. Taha and T. Sheard. Multi-Stage Programming with Explicit Annotations. *ACM SIGPLAN Notices*, 32(12):203–217, 1997.

[14] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. In *Theoretical Computer Science*, pages 203–217. ACM Press, 1999.

[15] A. M. Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[16] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java Multi-stage Programming Using Weak Separability. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 400–411. ACM, June 2010.