Building Code with Dynamic Staging

Piotr Danilewski\textsuperscript{1,2,3} Philipp Slusallek\textsuperscript{1,2,4}
\textsuperscript{1}Saarland University, Germany \textsuperscript{2}Intel Visual Computing Institute, Germany \textsuperscript{3}Theoretical Computer Science, Jagiellonian University, Poland \textsuperscript{4}Deutsches Forschungszentrum für Künstliche Intelligenz, Germany
piotr.danilewski@tcs.uj.edu.pl slusallek@cg.uni-saarland.de

Abstract

When creating a new domain-specific language (DSL) it is common to embed it as a part of a flexible host language, rather than creating it entirely from scratch. The semantics of an embedded DSL (EDSL) is either given directly as a set of functions (shallow embedding), or an AST is constructed that is later processed (deep embedding). Typically, the deep embedding is used when the EDSL specifies domain-specific optimizations (DSO) in a form of AST transformations.

In this paper we show that deep embedding is not necessary to specify most optimizations. We define language semantics as \textit{action functions} that are executed during parsing. These actions build incrementally a new, arbitrary complex program function.

The EDSL designer is able to specify many aspects of the semantics as \textit{action functions} that are executed during parsing. These actions build incrementally a new, arbitrary complex program function.

The EDSL designer is able to specify many aspects of the semantics as \textit{action functions} that are executed during parsing. These actions build incrementally a new, arbitrary complex program function.

The EDSL designer is able to specify many aspects of the semantics as \textit{action functions} that are executed during parsing. These actions build incrementally a new, arbitrary complex program function.

The EDSL designer is able to specify many aspects of the semantics as \textit{action functions} that are executed during parsing. These actions build incrementally a new, arbitrary complex program function.

The EDSL designer is able to specify many aspects of the semantics as \textit{action functions} that are executed during parsing. These actions build incrementally a new, arbitrary complex program function.

1. Introduction

Creating a new DSL from scratch is not an easy task. Even with the help of parser generators, such as YACC [Joh75], the developer still needs to handle several compiler-related tasks such as creating and transforming an abstract syntax tree or implementing variable lookup rules or a type system.

An alternative approach is to \textit{embed a DSL} – define it as part of a flexible, general-purpose \textit{host language}. The DSL is defined by a set of functions and other constructs giving the user a perception of programming in a new language.

In this paper we focus on \textit{action code} – pieces of code that define the semantics of the language. The simplest approach for the action is to contain the intended domain program behavior directly. When the associated syntactic construct is encountered by the interpreter or parser, the domain program is executed immediately.

Alternatively, one uses some form of \textit{staging} – a user-guided partial evaluation mechanism where the programmer annotates the code, guiding the interpreter to execute the code out of the normal order. In the context of a DSL definition, staging is used to defer the execution of the domain program. Most typical approaches of staging, such as Lightweight Modular Staging (LMS) [RO12], produce a program as an Abstract Syntax Tree (AST). The generated program, represented as an object, can be then traversed, analyzed, optimized and later compiled and run separately.

One embeds a DSL to simplify the definition creation, and to reuse code between the host language and the DSL. However, when the code is represented as an AST these traits are lost. The AST traversal and transformations must once again be defined. Moreover, the host language can no longer be seamlessly integrated with the produced program: The host code cannot be directly reused in the domain and the implementation details of the embedding can become visible to the end user [Jov+14].

Contributions

In this work we show a novel way to define language semantics without representing it as an AST object and traversing it separately. Despite that constraint, the language should still be able to produce highly efficient code.

We create \textit{fragment functions} which conceptually act as tree nodes. Unlike existing solutions, however, their behavior is defined entirely by their body, without any external traversing algorithm. A single node can contain arbitrary code and realize a complex language logic. We show:

\begin{itemize}
  \item How to define and connect these fragment functions.
  \item How to use staging to remove overhead and specify early-stage optimizations.
  \item How to define tasks which are typically realized through AST traversing algorithms. This includes basic operations, such as variable name lookup rules, as well as more advanced domain-specific optimizations.
\end{itemize}

Our solution has simplicity of shallowly embedded DSL and the power of deep embedding.
### 2. Related Work

#### 2.1 Staging

We call **staging** any method that allows symbolic computation “under a lambda”, that is: performing $\beta$-reduction within bodies of functions that have not been invoked yet. We define the following types of staging:

The simplest, most crude approach is **textual staging**, where a program is composed from strings containing pieces of source code. Textual staging provides no correctness guarantees about the generated program and has limited mechanisms for communication between the stages.

In **structural staging**, the program code is represented explicitly as a data structure, typically as an AST or graph. The structure can be created explicitly, for example through the LLVM instruction builders. Alternatively, the process can be hidden behind overloaded functions, operators, or templates [Vel96]. Data exchange between stages is inconvenient since it requires conversion from values to structure nodes and vice-versa. Moreover, nested staging requires a structure to describe a program building another structure, making such approach inconvenient to use.

**Functional staging** is a case of structural staging where the structure is represented by ordinary functions [Rey75; CKS09]. Such functions can be combined to form bigger pieces of code. The piece of code is executed by simply invoking the function representing it. In Lightweight Modular Staging (LMS) [RO12] these builder functions are hidden by overloading ordinary functions over a higher-kind type $\text{Rep}[\tau]$. Consequently, the staged code does not differ much from the normal code. Unlike earlier approaches, communication is facilitated through an implicit promotion of values from $\tau$ to $\text{Rep}[\tau]$ where needed. Still, the structural staging is not entirely hidden and sometimes it leaks to the programmer. As a result, they must be aware of the internal usage of staging within the builders [Jov+14].

#### 2.2 Language Embedding

Languages can be created as standalone products, or as a library in a flexible host language such as Haskell, ML, or Scala. The second approach is referred as **language embedding** as it defines the DSL within the *host language*.

Such an approach reduces the complexity of the language creation. Most of the compiler-specific challenges, such as having a sound type system, are already resolved in the host language and can be reused in the DSL. The author can focus only on aspects that are truly unique for the domain.

Having the common base language, DSLs can use it to relay information between the domains and the host language. For this reason, this approach is common when defining small, embedded DSLs (EDSLs). For example, Haskell has been used extensively to define geometric operations [HJ94], CAD operations$^2$, COM component scripting [JML98], hardware design [Bje+98], or server-side web scripting [Mei00; Thi05].

### Figure 1: The relation between different staging methods and language creation types.
Basic syntax: \[
\lambda[y][x].b
\]
\[
\top \downarrow \text{ and or not}
\]
\[
[e]v\bar{r}
\]
\[
[e]\text{fix}[y]x = v \text{ in } b
\]
Syntactic sugar: \[
\lambda[y][x].[y]\bar{v}
\]
\[
[e]\lambda(y)[x].b\ v
\]
\[
[e]v\bar{r}(\lambda[y][x].b)
\]
\[
(\lambda[y][x].b) \text{ expr} \xrightarrow{\beta} (\lambda[y][x].b) \ v \text{ expr}(x)[y] \ b
\]

**Figure 2:** Comparison between lambda calculus extended with dynamic staging (left), and the actual syntax of DeepCPS (right).

Most EDSLs focus on semantics, while inheriting the syntax of the host. The semantics is defined by pieces of code, that we refer to as actions. Depending on its contents we distinguish two forms of embedding:

- **In shallow embedding** the associated semantics is represented directly in the action and is executed immediately when the syntactic construct is encountered.
- **In deep embedding** a program structure representing the intended behavior is created, that can later be translated, optimized, and run separately.

This distinction is closely related to the form of staging used within the actions. Typical shallow embedding uses no staging and the code is executed together with the embedded syntax representing it. Deep embedding on the other hand uses structural staging to generate a program based on the syntax. Direct structural staging is still very common, as in Haskell/DB [LM99] or Scala [HO10]. However, more advanced staging approaches become popular in context of EDSLs as well, using LMS or MetaML [Cha+10; Ofe+13; SBP99].

This relation between the type of staging and the depth of the embedding, as depicted in Figure 1, are of particular interest to us. In our work we provide a solution which, due to the flexibility of staging, is on the verge between what can be considered shallow and deep embedding.

### 3. Dynamic Staging

Before we present our main work let us quickly summarize the dynamic staging and DeepCPS language as our work heavily utilizes it. The full information, including the formal definition, can be found in [Dan+14].

#### 3.1 DeepCPS

DeepCPS is a functional language operating entirely in the Continuation Passing Style (CPS) [App07]. In Figure 2 we summarize the syntax of DeepCPS. The basic lambda value \( \lambda[\bar{r}].b \) is represented as \( (x_1, x_2, \ldots) \{y\}[b] \). If the lambda is the last argument in an application – a scenario common in the CPS programming – the braces surrounding the body \( b \) can be omitted. This way DeepCPS avoids the deep nesting of parenthesis, typically present in other CPS representations.

Dynamic staging is introduced through the implicit staging parameter \( [y] \) that is present in every lambda and a staging function \( \text{fix for}(!\text{args}, \text{startval}, \text{endval}, \text{body}, \text{endfor}) \{ \text{startval}!\text{endval}(\text{cond}) \}

- If \( \text{cond} () \) then \( \text{body} !\text{args} \text{startval}!\text{endfor} \{!\text{args2} \text{startval}!1(\text{nextval}) \}
- For \( !\text{args2} \text{nextval}!\text{endval}!\text{body}!\text{endfor} \)
- \( () \)
- \( \text{endfor} !\text{args} \)

**Listing 1:** A for loop defined as a higher-order recursive function, using the DeepCPS syntax. In each step, the user-provided body function is called, passing in and out a tuple of arbitrary arguments \( !\text{args} \). Upon reaching the end, the endfor continuation is invoked.

- **expression \( @e: \)** in each lambda body. When a lambda is invoked, the implicit staging parameter is substituted by a staging constant \( \top \). Staging expressions use these staging parameters to form boolean expressions. When \( @e: \) evaluates to \( \top \), the annotated body is considered active and is scheduled for the execution. At the each execution step the deepest active body, containing no nested active bodies, is executed.

Staging variables can be used as normal arguments, and normal variables can appear in staging expressions as well. Non-stage constants are equivalent to \( \bot \), while variables that are still represented only as symbolic values are \( \top \).

As syntactic sugar, the staging expression can be omitted, implicitly staging the body on the lambda it is directly contained in. When that lambda is invoked, the body is executed as if no staging was present. We refer to this as natural staging.

#### 3.2 Control Flow

Control flow in DeepCPS is realized through a single built-in function \( \text{if taking 3 arguments: cond, tb, fb}. \)

Other flow functions can be defined with the help of \( \text{if} \), for example as shown in Listing 1. The \( \text{for} \) function recursively calls itself, incrementing the initial value \( \text{startval} \) by 1 in each call. In each iteration, the body is invoked with the current value of \( \text{startval} \). The recursion is terminated when the \( \text{endval} \) value is reached.

A parameter preceded by \( ! \) accepts any excessive arguments given to a function, and packs them all into a tuple.
We assume that each action has means to communicate with args with 6 arguments, the first two are put into the tuple. The most straightforward approach is to provide the intended communication is performed in one direction, in the order other actions by passing arguments. We only require that the functions, or as an embedding in another language.

Consider a language designer creating a new DSL. Let us assume that the grammar for such language already exists. The precise representation of the grammar does not interest us: the grammar may be defined in a standalone parser, in a Backus-Naur form [Bac+63], or through a set of parser combinator functions, or as an embedding in another language.

The grammar contains semantic actions in a form of action builder functions. We introduce builder functions — a CPS approach for such encapsulation, consisting of 3 layers of functions:

• The core constitutes a piece of code that we want to generate. In CPS representation, it is a function taking any number of arguments, performing its user-defined function and then returning through the continuation with a (possibly different) set of arguments.

• The core is encapsulated by a fragment function which specifies the continuation for the core.

• Finally, a builder function acts as a constructor: When invoked with an arbitrary code as an argument, it creates a fragment function with the core containing said code. A simple builder from Listing 5 calls its continuation return with a newly created fragment function. By invoking

under the given name. When, for example for is invoked with 6 arguments, the first two are put into the tuple args.

3.3 Staging Chain
One of the benefits of DeepCPS that we rely on in this paper is the ability to form staging chains. A staging chain is a sequence of staging variables $s_1, s_2, s_3...$ where $s_i$ stages a piece of code that is ending with a continuation defining $s_{i+1}$. The simplest case is shown in Listing 2, but these pieces can originate from completely independent functions. A single piece may be encapsulated in a function such as:

$$
\text{Listing 2: The staging chain pattern. When } s_1 \text{ becomes active, code 1 is executed. Its final continuation activates } s_2, \text{ triggering the execution of the next code piece. This way, all the code pieces } code_n \text{ are executed in sequence. The code pieces can occur at arbitrary locations in the program, even in separate functions, as long as the staging values that form the chain are properly passed.}
$$

under the given name. When, for example for is invoked with 6 arguments, the first two are put into the tuple args.

The symbol ! in front of an argument unpacks all elements of a tuple and splices them as an argument list to a function call. For convenience, all arguments that are packed or unpacked through !, are highlighted in italics.

4. Building Code
Consider a language designer creating a new DSL. Let us assume that the grammar for such language already exists. The precise representation of the grammar does not interest us: the grammar may be defined in a standalone parser, in a Backus-Naur form [Bac+63], or through a set of parser combinator functions, or as an embedding in another language.

The grammar contains semantic actions in a form of action functions, executed when the given grammar rule is taken. We assume that each action has means to communicate with other actions by passing arguments. We only require that the communication is performed in one direction, in the order that the actions are invoked. This can be the order of parsing, e.g. by using an L-attributed grammar [LRS74].

The question we answer in this section is how to specify these actions, so that by invoking them we generate a new program code. We use the DeepCPS syntax, but for the moment use no staging.

4.1 Immediate Execution
The most straightforward approach is to provide the intended semantic meaning of the parsed code directly in the action. Suppose for example, one implements a simple stack-based interpreter of binary arithmetic expressions as in Listing 4:

$$
\text{Listing 3: The push and pop functions for a stack represented as a nested tuple } \langle v, tail \rangle. \text{ Each element } v \text{ is paired with a tail representing the rest of the stack. After each operation, an updated stack is passed into the continuation return.}
$$
In section 4.3, we discuss how to fix this problem using staging. Without staging, we actually obtain a result that is only \( \beta \)-equivalent to the one shown above. Working with such fragments would incur a major overhead. We fix this problem using staging in Section 4.3.

The resulting fragment function has the same structure as a fragment function created by the builder, and can be further merged with other fragments to build bigger programs.

We must remark however, that the reduction described above is performed within the body of the fragment lambda returned by the merge function. This implies staging. Without it, we actually obtain a result that is only \( \beta \)-equivalent to the one shown above. Working with such fragments would incur a major overhead. We fix this problem using staging in Section 4.3.

Listing 5: The basic builder and merge functions for our functional code building. The builder creates a fragment function that encapsulates a piece of arbitrary user code, given as an argument to the builder. The merge function connects two existing fragment functions \( F_{prev} \) and \( F_{next} \), forming a single new fragment containing all user code from \( F_{prev} \) and \( F_{next} \).

Listing 6: Semantic actions for an interpreter of arithmetic expressions, creating a lambda function with the code containing all the parsed operations.

With the help of the builders we now redefine our arithmetic expression language. In Listing 6, we no longer execute the code immediately. Instead, each of the actions creates a new fragment, and merges it with the previous one (\( F_{prev} \)).

4.3 Removing the Builder Overhead

The simple builders we presented so far are lazy in their evaluation: the merging of the fragments does not occur until the actual program is executed, incurring a potential major overhead. Instead, we would like to produce any code as if it was written directly in DeepCPS. The whole layer of builders, fragments, and core functions calls should resolve early, leaving only the user-defined code. Moreover, we would like to be able to express optimization and staging strategies in the semantic actions as well.

In this section we distinguish three execution phases:

- The build-time (\( bt \)) when the builders are being invoked.
- The core-time (\( ct \)) when all code pieces are already merged into a single fragment and are ready to be transformed into a produced program.
- The program-time (\( pt \)) when the function that we created – referred as a program – is invoked.

We avoid the more common terms compile-time and runtime, because this distinction can be misleading in the context of our builders and dynamic staging. This is because all the execution occurs in the very same language: DeepCPS. All code, including the programs we construct, can be executed immediately, even during build-time of another function. Thus, a program-time for one function can be a build-time for another. Such flexibility becomes lost when thinking in rigid terms of compilation and final run-time execution.

With the dynamic staging it is possible to achieve our goals. The staged builders in Listing 7 have the same structure as the original ones from Listing 5. The difference is the use of staging within the bodies of the build and merge functions.

In Listing 8 we show how the staging allows the merge to remove the overhead of having layered fragment function calls. Suppose that merge is invoked with two fragments \( F_{next} \) and \( F_{prev} \) produced by the builder. As part of the invocation, the implicit staging parameter \( bt \) becomes \( \top \) and triggers immediate execution of the code within the body of the merge fragment (Listing 8b).
Step 3: The values replace the symbolic names in the universal merge function.

Listing 8: Example run of staged `merge` invoked with fragment function arguments produced by the staged `build`.

![Listing 8](image)

The `merge` is given a symbolic argument `next`. The function merely invokes the returning lambda continuation, with `(ct, pt, !args) ...` as the body argument (Listing 8c).

This is followed by the invocation of `Fprev`. This time however, the fragment argument `next` is a concrete lambda – the previously returned core of `Fnext`. As a result, within the body of the fragment `Fprev`, the `next` function is immediately invoked. Consequently, the whole line:

```
@next: next ct pt !args2
```

is replaced with the core of `Fnext`. Only when that happens, the fragment `Fprev` returns (Listing 8d).

Suppose that we have constructed and merged all fragments representing our program. What we get is a single fragment function of the form:

```
let Fcomplete(next, return) { return
  (ct, pt, !args)
  @ct: code1 pt !args (pt !args2) [ct2]
  @ct2: code2 pt !args2 (pt3 !args3) [ct3]
  @ct3: code3 pt !args3 (pt4 !args4) [ct4]
  ...
  @next: next ctn ptn !argsn
}
```

We can identify the use of the fragment chaining pattern from Section 3.3 twice, associated with `ct` and `pt`.

The core-time staging chain `ct` is used to stage the calls to the user functions `code_n`, effectively executing the contents of the core functions (hence the staging name). This resolves the process of packing and unpacking parameters into the `!args` tuples. After the core-time chain is resolved, only the code actually provided by the user remains.

If the user code has no staging (that is – has only natural staging), all of it is evaluated at core-time. However, by using the program-time `pt` staging chain, execution of parts of the code are delayed and spliced into `Fcomplete`. The code staged upon `pt` constitutes the program we generate.

Notice that the builders themselves never stage upon `pt` – it is merely passed around together with `!args`. In fact, `pt` can be part of `!args`, but we included it explicitly for clarity. Moreover, `!args` can include more user-defined staging variables. With these, users are able to define arbitrary additional dynamic staging within their code.

4.4 Branching via Universal Builders

The basic builders we work with so far assume that the code requires a single continuation. We say that these fragment functions are of arity 1. However, when more continuations are needed, the signature of the fragment changes and the
let build

(arity, code, return)[bt] { return

    let builder

    [arity, code, return][bt] { return
    
        @bt: for Snext () 0 arity
        (Snext, Tlambda, i, break, continue)[bt] { @next: pop Snext (next, Snext)
        @bt: let lambda_i
        (pt, !args2)[ct]{ continuation λ_i
        @next: next ct pt !args2
        }
        concat(Tlambda, [lambda_i]) (Tlambda) continue Snext Tlambda
        }
        (Snext, Tlambda) //endfor
        push Snext
        (ct, pt !args){
            @ct: code pt !args !lambda Tlambda
        } (S)
        @return: return S
    }

    let merge

    [prev, fnext, return][bt] { return

        @bt: Fprev Snext1 (Snext2)
        Fprev Snext2 (Snext3)
        @return: return Snext3
    }

Listing 9: The universal builders handling fragments of arbitrary arity. The fragments exchange a stack of continuations Snext, rather than a single continuation next. In the for loop, the fragment takes as many elements from the stack as needed, defined by the arity parameter, and pass them into the user-defined code as continuations.

Builders have to be adjusted. For example, when encapsulating a condition function if that takes two continuations:

let build2 (trueCont falseCont) {
    if cond trueCont falseCont
}

one needs to use a version of the builder which creates a fragment function of arity 2, for example defined as build2:

let build2

(code, return) { return

    (next1, next2, return) { return

        (!args)
        code !args (!args2) { next1 !args2 }
        (!args3) { next2 !args3 }
    }
}

Fragment functions of different arity present a problem as they cannot be directly used with the merge function presented earlier. On the other hand, creating a merge function to match fragments of different arity would be ineffective – the user would have to use several versions of the function, where the distinction could be automated.

Instead of using specialized builder and merge function, in Listing 9 we introduce a universal solution for any arity.

The main difference is that we no longer pass a single next continuation into the fragment functions. Instead, we pass a stack Snext, containing arbitrary many continuations. We use the same stack implementation as previously in Listing 3.

When creating a new fragment through the builder, the user specifies the intended arity. This specifies the number of continuations passed by the builder to the user-provided code that follow the standard !args. These continuations are kept in the Tlambda tuple, created at build-time in the for loop at the beginning of the builder’s fragment function. Each of the next values used by these continuations is obtained from the Snext stack as soon as it is available.

The universal merge function does not differ much from its simple predecessor. However, this time we do not pass a single core argument between the fragments, but the whole continuation stack Snext. The code within the universal builder, through the use of lambda_i performs the actual merging. The merge function merely specifies which two fragment functions are connected.

Note that the push operation on the stack is performed in build-time even if Snext is a symbolic value. This allows us to create stack with top element known, and unknown remaining tail. This can be seen for example in Listing 10. Such element can be later used in another fragment to resolve the merge, even if the whole stack is not provided.

The logic of the construction through these builders is from back to front: when the continuations representing the ending part of the program are known, they are put on the stack and can be merged with a new fragment of code preceding it. Despite that it is not necessary to call these functions in that one particular order. Thanks to the operations on incomplete stack, the fnext can be successfully merged to fprev even if the former is just a short piece of code without any known continuations.

5. Examples

In the previous section we have shown how arbitrary code can be created using a functional approach. We have explained how dynamic staging removes the overhead of the layered construction. Let us now show how the builders can be used in practice.

5.1 Arithmetic Expressions

At the beginning of Section 4, in Listing 4 we provide semantic actions for parsing arithmetic expressions. In there, however, all the computations are performed immediately during parsing. With the help of the builders, the complete set of semantic actions would look like in Listing 11.
Let `start(return) {` build 1 { `cont { cont `exit [] } } return } let `number(Fprev, val, return) {` build 1 { `cont `exit `push `cont `pop `cont `push `cont `exit } (F) merge Fprev F return } let `add(Fprev, return) {` build 1 { `cont `exit `pop `cont `push `cont `exit } (F) merge Fprev F return } let `end(Fprev, return) {` build 0 { `cont `exit `pop `cont `exit } (F) merge Fprev F return } let `sub` `mul` `div` `end(Fprev, return) {` build 0 { `cont `exit `pop `cont `exit } (F) merge Fprev F return } Listing 1: Semantic actions for an interpreter of arithmetic expressions. Each action builds a fragment with the provided user code. The user code employs a number stack to accumulate intermediate values. The ending action “seals” the chain with a fragment of arity 0, taking no more constructed continuations.

The `start` action is executed first, at the beginning. Apart from the continuation, it takes the function-time staging variable, and the `exit` continuation that should end the program we are building. Moreover, we initialize the number stack with an empty value `[]`.

The `number` and `add` actions construct a fragment of arity 1, similarly to what was done in Listing 6. All operations on the number stack are performed at core-time, even if the numeric values, such as `sum` are not known at that time. Only the actual addition is delayed to be part of the program we generate, executing at program-time.

Finally, in the `end` semantic action we seal the fragments. That means, we provide the last fragment with arity 0, which calls `exit` to end the program. The fragment representing the complete program is invoked with an empty stack as `Snext`, returning a `program` which is a naked user code without any encapsulation. By invoking the program under the lambda `(exit)[pt]...` we trigger the core-time stage, leaving only the code staged upon `pt` in the body of that lambda.

The presented approach using a number stack is not the only way to realize an arithmetic expression semantics. Alternatively, the fragments can be connected in an AST-like fashion as in Listing 12. The action associated with the operator creates a fragment function of arity 2. Each child fragment represents a subexpression on the left and right side of the operator, respectfully. A literal number, on the other hand, acts as a leaf in the tree of fragments and has arity 0.

In both approaches, we rely on the language grammar to invoke the semantic actions in the correct order. In particular, the grammar handles the precedence of the operators.

Regardless of whether we use a number stack, or AST-like fragments, we obtain the same end result. This is because the merging and calls into each user code function is resolved early. Only instructions in the `pt` chain remain. They are the same in both approaches. If, for example, we try to parse `1+4*2+3`, the produced program for both cases is exactly:

```plaintext
(exit)[pt] { Bpt: exit sum Bpt: 4+2 (prod)[pt] Bpt: I+prod (sum)[pt] Bpt: sum+3 (sum)[pt] Bpt: exit sum }
```

5.2 Custom Environment

Suppose that a DSL requires name binding. With functional code building we do not have to rely on a single approach or follow the rules of the host language. Name binding can be handled explicitly by the user code, possibly in an early stage to avoid program-time overhead.

In order to manage the names, we need a map `M` that maps string names to values. We do not care how `M` is
All map operations are performed at core-time, allowing the
Typically, an optimization is specified as a separate pass over
functionality and user code functions. The insertion and lookup is performed at build-time,
leaving no trace of it at function-time.
implemented. It may be implemented in a purely functional
manner, with each operation producing a new \( M \) value. It may use stateful memory operations. It may even use
external C functions for efficiency purposes. It may also feature a mechanism to reference other maps to support
nested scopes. Since dynamic staging is explicit and does not rely on program analysis, any implementation can be
used with our approach.
We only require that \( M . \text{insert}(\text{name}, \text{value}) \) should bind, possibly symbolic, \text{value} to \text{name} in the map \( M \), while \( M . \text{lookup}(\text{name}) \) should return the previously stored \text{value}.
As an example, we can now augment our arithmetic expression language to support name binding, as in Listing 13. All map operations are performed at core-time, allowing the map itself to be removed before the program-time.
This is the simplest example of an auxiliary computation that can be added into our functional code builders. Such computation does not constitute the main program logic. Instead, it is an additional support code that can be used for management, verification or domain-specific optimization.

5.3 Domain-Specific Optimization

Typically, an optimization is specified as a separate pass over the program structure. In such a pass, characteristic patterns are found and the AST is transformed, often violating the inherent node semantics.
With the functional builders and dynamic staging the optimization can often be specified as a function as well. Consider for example our arithmetic expression language operating on matrices of arbitrary dimensions. Each matrix \( M \) is represented by its dimensions \( M . \text{dim} \) known statically, and its contents \( M . \text{data} \) that are known only at program-time.
Consider a problem where a sequence of matrices has to be multiplied. Finding the optimal sequence of multiplications is a classic problem that can be solved through dynamic programming [Cor+09, Chap. 15.2]. Suppose that the algorithm is implemented in a function \( \text{matrix_chain_order}(d) \),

Listing 13: Extending the AST-based interpreter of arithmetic expressions to support name binding. The parameter \( M \), acting as a binding map, is passed through all user code functions. The insertion and lookup is performed at build-time, leaving no trace of it at function-time.

Listing 14: Chain matrix multiplication as an example of domain-specific optimization. The \text{mulNext} action collects all the matrices, but the classic algorithm is contained entirely within the \text{mulEnd} action. There, the optimal order is found in \( \text{matrix_chain_order} \). Based on the result, matrices are multiplied in a for loop. All the optimization is performed in the core-time staging chain. At program-time the loop is unrolled and only the multiplication operations remain, referencing directly the right matrices.

which for given tuple \( d \) of matrix dimensions computes a sequence describing the optimal multiplication order. As before, the implementation of such function does not concern us.

In Listing 14 we show how such optimization can be specified in the semantic actions. The \text{mulStart} is invoked once at the beginning of the sequence, initializing the list of matrices \( Ms \) with an empty tuple. The \text{mulNext} is used for every member of the matrix chain and accepts 2 continuations. The \text{child} represents a matrix subexpression yielding a single result that needs to be multiplied. The \text{continue} refers to the remainder of the matrix chain.

Finally, at the end, in \text{mulEnd} we have all the data to launch the algorithm. The dimensionality information is extracted from \( Ms \) and the optimal multiplication order is found. With each iteration of the for loop a single line of code is produced, staged for program-time, which multiplies
the matrices selected by the optimal order. All other support code: launching the algorithm, managing the matrix tuple and computing the dimensions of the result are executed early, at core-time.

6. Discussion
Let us discuss how our functional approach for code building relates to both the shallow and the deep language embeddings.

Having the semantic action, containing the intended program logic represented as a function is typical for shallow embedding. Every host language function can be used directly in the DSL and vice-versa, any program constructed by the DSL can be used in the host. This is in contrast to deep embedding where host programs must be converted to an AST representation. Such conversion is difficult to achieve transparently especially when the built-in functions and constructs, such as conditionals, are encountered.

Typical shallowly embedded DSLs have an overhead stemming from the additional layers of functions tied to the language syntax. As we have shown in Section 4.3 however, dynamic staging can resolve all overhead coming from the embedding, producing only the intended user code. Moreover, unlike typical shallow embeddings, the produced code is not directly tied to the DSL syntax. The actual computation may occur later or even in a different order, as in the chained matrix example in Section 5.3. It is also possible for a DSL to produce not one but two or more chains of fragment functions. Such technique can be used to generate a multi-pass interpretation in a single parsing, e.g. where circular definitions are present.

On the other hand, our functional builders resemble the deep embedding as well: Actions produce code objects and the code is built incrementally, by connecting the fragment functions in a tree structure similar to an AST.

Nodes in our graph structure are self-descriptive, i.e. their behaviour is entirely defined within the functions they contain. As shown in Section 5.2 these functions may contain much more than just the core semantics. The user may define many staging phases and incorporate additional, auxiliary computation within it. It may be used to define custom name lookups, but also as an analysis step, optimization, or even perform custom type checking [Her10].

Unlike the deep embedding however, we do not support tree transformations. Optimization is not achieved through analysis of the program graph, but through the partial evaluation enabled by dynamic staging. We do not claim that it is the only way programs should be optimized: careful analysis of the code is needed to generate fast, low-level machine code. What we claim, instead, that such low-level techniques are usually not needed at higher-level DSL definitions and generally should be avoided because of their complexity.

This paper focuses on the conceptual aspects of building code through dynamic staging. However, it is important to state that the presented approach has been fully implemented in the DeepCPS interpreter and compiler. This includes the examples presented here, as well as more complex DSLs, such as a language for grammar specification and for stencil code generation. In all the cases we were able to specify domain optimization, the interpreter was able to remove any residual overhead and the compiler produced an efficient machine code.

7. Conclusions and Future Work
We have presented a novel approach to code building. We use an entirely functional approach, where each builder produces a fragment function containing an arbitrary program code. We have shown how builders are defined and argued how dynamic staging allows us to remove the overhead coming from such encapsulation.

In Section 5 we have given simple examples of how typical language design problems can be handled using builders and dynamic staging. This includes domain-specific optimizations, which are also represented as additional code. It is more natural for a domain-expert to implement an optimizing algorithm as a program, rather than as an AST transformation. The whole optimization algorithm can be hidden away in a single builder node, or even provided as a separate function, independently of the DSL definition.

We did not include the performance numbers for the machine code produced by our DSLs as we do not evaluate the efficiency of the DeepCPS compiler. Such evaluation has been done in [Dan+14]. Note, that we are able to produce the DeepCPS code containing no unnecessary residual operations originating from the builders. The back-end compiler receives the same input as if the code was written without the builders, and produces the same result.

Future Work
The focus in this paper is only on the semantic actions of a language. For a fully functioning DSL this is not enough – a parser or an embedding is needed. We did not provide any integration of our semantic actions with a parser, nor explained how to embed a DSL in DeepCPS. Such integration is in our current focus and we have made substantial advancements in that area. This is however not in the scope of this paper.

In Section 5.2 we described how dynamic staging can be used to define an auxiliary computation. The same approach can be used to build custom type systems. In theory, any type system can be defined as a staged computation and used in a DSL. To our knowledge this possibility has not yet been fully explored and requires further research. This may be particularly interesting in the context of DeepCPS as dynamic staging may bring dynamic and static typing closer together, in a similar way as it did for shallow and deep embeddings.

We also hope to explore other uses for functional code building, not directly related to language definitions. As each fragment can be an arbitrary large piece of code, the same technique can be used to build programs at bigger granularity. For example, this technique may become viable for data-flow programming, and semi-automatic, profiled assembly.
