Funcons for HGMP

The Fundamental Constructs of Homogeneous Generative Meta-Programming

L. Thomas van Binsbergen
Department of Computer Science
Royal Holloway, University of London
ltvanbinsbergen@acm.org

Abstract
The PLanCompS project proposes a component-based approach to programming-language development in which fundamental constructs (funcons) are reused across language definitions. Homogeneous Generative Meta-Programming (HGMP) enables writing programs that generate code as data, at run-time or compile-time, for manipulation and staged evaluation. Building on existing formalisations of HGMP, this paper introduces funcons for HGMP and demonstrates their usage in component-based semantics.

ACM Reference Format:

1 Introduction
The PLanCompS project\(^1\) proposes a formal, component-based approach to programming language development. The aim is to reduce the initial effort of writing formal specifications and of maintaining the specifications as languages grow by reusing components across specifications.

Funcons Central to the approach is a library of highly reusable ‘fundamental constructs’ called funcons. Funcons are not altered after their release, thereby fixing language specifications that depend on them. The beta version of the funcon library is available online for review \([18]\).

Funcons have been identified for many aspects of programming: functions and procedures, references and mutable storage, scoping and binding, patterns and pattern-matching, as well as exceptions, delimited continuations and other forms of abnormal control-flow \([9, 20]\).

Funcons are formal and executable: each funcon has operational semantics and interpreters are generated from their definitions \([5]\). A language is defined formally by a translation of programs to ‘funcon terms’. A language definition is tested by translating programs to funcon terms, executing the funcon terms, and comparing the observed behaviour with the desired behaviour. This paper assumes some familiarity with the funcon approach, of which an overview is given in \([9]\).

Funcons for HGMP A language with constructs for Homogeneous Generative Meta-Programming (HGMP) enables writing code that generates code. As data, the generated code can be propagated and manipulated freely, before being inserted and evaluated in the overarching program. Template Haskell \([21]\) supports HGMP at compile-time, MetaML \([22]\) at run-time, while Converge \([23]\) supports both. An overview of the features of several HGMP languages is found in \([7]\).

In this paper, we define funcons for HGMP, raising several research questions. Can we use the funcons for HGMP in component-based semantics? What is their coverage? Are they sufficient to give semantics to many real-world and academic languages? Can we implement them such that translations and funcon terms that use them are executable? This paper answers the first question by demonstrating the usage of the funcons for HGMP in component-based semantics.

Section 2 introduces a standard \(\lambda\)-calculus as the running example. Section 3 defines the funcons for HGMP. Section 4 adds HGMP constructs to the \(\lambda\)-calculus and gives a component-based semantics based on the funcons for HGMP. Section 5 shows that the funcons for HGMP enable a straightforward semantics for call-by-need (lazy) evaluation.

2 Component-Based Semantics – Example
This section introduces the running example of this paper, a component-based semantics for a call-by-value lambda-calculus \(\lambda_v\). We have chosen a lambda-calculus with standard and well-known call-by-value semantics. This allows us to focus on the method for specifying the semantics, rather than the semantics itself. We use the funcons of the beta release to specify the semantics. For an intuitive understanding of their behaviour, we refer the reader to the online
we assume programs are well-typed. The function returned by a lambda-expression \( \lambda x.e \) is statically scoped by computing a closure (rather than using a different approach)

2We assume programs are well-typed.
Funcons for HGMP

In this section we identify funcons for HGMP based on formalisations of HGMP by Berger and Tratt [7, 8]. In [7], Berger, Tratt and Urban present a calculus for reasoning about several aspects of HGMP. Their calculus is the result of applying a semi-mechanical ‘HGMPification’ recipe to a standard untyped λ-calculus, similar to λV. The recipe extends languages with abstract syntax trees (ASTs) — to serve as meta-representations of program fragments — and several HGMP constructs. Here, we define funcons for building ASTs and a funcon for most of the constructs added by the recipe.

We apply the HGMP recipe to an unspecified set of funcons C, making several assumptions about C. We assume a distinction between values and computations, where a term f(t1, . . . , tn) is a value if and only if f is in some subset CV of C. A constructor in CV is referred to as a value constructor, a constructor in C \ CV as a computation constructor, and a non-value term as a computation. This distinction is important as values are assumed to be fixed: they have no computational behaviour and they have the same meaning wherever they appear. Similarly, we assume that some values are types, i.e. a value f(t1, . . . , tn) is a type if f is in some subset CT of C. A constructor in CT is referred to as a type constructor, a constructor in C \ CT as a type constructor, and a non-type term as a type.

Following [9, 18], we express the semantics of the funcons for HGMP using I-MSOS rules [16], a variation on Modular Structural Operational Semantics (MSOS) rules [15] in which so-called ‘auxiliary semantic entities’3 are implicitly propagated. The I-MSOS rules for funcons that do not interact with semantic entities are indistinguishable from conventional Structural Operational Semantics rules [19], and only the meta-let funcon for compile-time meta-programming actually interacts with a semantic entity.

We assume that all computation constructors are associated with small-step I-MSOS rules defining the relation ↓. Finite evaluations are captured by the ‘iterative closure’ ↓ →, of ↓, expressing that c evaluates to value v when c ↓ v. The iterative closure is defined as:

\[ f \in C_V \quad f(t_1, \ldots, t_n) \rightarrow f(t_1, \ldots, t_n) \quad (2) \quad c_1 \rightarrow c_2 \quad c_2 \rightarrow \cdots \quad v_1 \quad (3) \]

Abstract syntax trees We add the type asts and the value constructor astv, for constructing ASTs representing funcon terms. There are two types of AST nodes. Firstly, an AST node can be labelled with a value v and a type τ, in which case it has no children. Secondly, an AST node can be labelled with a funcon f and have zero or more children. Funcons themselves are not funcon terms, only applications of funcons are.

To represent funcons, we add a (nullary) value constructor — referred to as a tag — for each computation constructor f, denoted by tag(f). ASTs are formalised by the following rules.

\[ \text{astv}(\tau, v) : \text{asts} \quad (4) \quad a_1 : \text{asts} \ldots \ a_n : \text{asts} \quad \text{astv}(\text{tag}(f), a_1, \ldots, a_n) : \text{asts} \quad (5) \]

Tags are necessary not only for the funcons in C, but also for the funcons for HGMP. We therefore introduce all funcons for HGMP simultaneously, deferring the explanation of their usage and semantics. The additional computation constructors are ast, code, eval, type-of, meta-up, metadown, and meta-let. The additional types are asts and tags.

The additional value constructors are astv and tag(f), with \( \text{tag}(f) : \text{tags} \), for each computation constructor f. Let C′, C′′, C′′′, be the extensions of C, CF, CV, and C T respectively, and let C′′′ replaces CV in Rule (2).

Meta-representation An AST is the meta-representation of a particular funcon term. The relation \( a \downarrow t \), introduced in [7] as \( \downarrow_a \), captures the conversion of a meta-representation a into the term t it represents. Relation \( \downarrow \) is defined for computations by the following rule:

\[ \text{astv}(\text{tag}(f), a_1, \ldots, a_n) \downarrow f(t_1, \ldots, t_n) \quad (6) \]

Variable f in Rule (6) ranges over computation constructors CF, which contains the unspecified set CT. For any particular CF, Rule (6) can be replaced by a collection of rules, one for each possible instantiation of f.

The following rule defines \( \downarrow \) for values:

\[ v' = \text{coerce}(v, \tau) \quad \text{astv}(\tau, v) \downarrow v' \quad (7) \]

Coercing v to a value of type τ may be necessary in a context in which values are paired with types at run-time. Otherwise, let \( v = \text{coerce}(v, \tau) \) for all v and τ.

The funcon ast constructs partially evaluated ASTs, e.g. give(true, ast(booleans.given)) requires evaluation to yield

---

3 Examples of semantic entities are stores (heaps) and environments, for modelling imperative storage and variable bindings respectively.
The dynamic semantics of \( \text{ast} \) is defined by the following rules:

\[
\begin{align*}
\text{ty}(v) &\Rightarrow \text{ty}(v) \\
\text{type-of}(v) &\Rightarrow \text{ty}(v) \\
\text{eval}(t) &\Rightarrow t' \\
\text{type-of}(t) &\Rightarrow \text{type-of}(t')
\end{align*}
\]

The HGMP recipe adds a construct for lifting values to their meta-representation. We decided to add \text{type-of} instead, which has applications outside of meta-programming, and show that lifting can be defined with \text{type-of} in Section 4.

---

\( Θ(\text{booleans, true}) \). The dynamic semantics of \( Θ \) is defined by the following rules:

\[
\begin{align*}
\text{eval}(\tau, v) &\Rightarrow \text{ty}(v) \\
\text{type-of}(v) &\Rightarrow \text{ty}(v) \\
\text{eval}(t) &\Rightarrow t' \\
\text{type-of}(t) &\Rightarrow \text{type-of}(t')
\end{align*}
\]

The HGMP recipe adds a construct for lifting values to their meta-representation. We decided to add \text{type-of} instead, which has applications outside of meta-programming, and show that lifting can be defined with \text{type-of} in Section 4.

---

\( \text{code} \) takes an arbitrary term \( t \) as argument and is dynamically replaced by the AST representation of \( t \):

\[
\begin{align*}
t &\Downarrow a \\
\text{code}(t) &\Rightarrow a
\end{align*}
\]

The funcon \( \text{eval} \) evaluates its argument to an AST \( a \) and is replaced by the term represented by \( a \).

\[
\begin{align*}
a &\Downarrow t \\
\text{eval}(a) &\Rightarrow t \\
\text{eval}(t) &\Rightarrow \text{eval}(t')
\end{align*}
\]

As an example, consider the evaluation\(^4\) in Figure 3.

The funcon \( \text{type-of} \) evaluates its argument to a value \( v \) and is replaced by the type \( \text{ty}(v) \).

\[
\begin{align*}
\text{ty}(v) &\Rightarrow \tau \\
\text{type-of}(v) &\Rightarrow \tau \\
\text{ty}(v) &\Rightarrow \tau \\
\text{type-of}(v) &\Rightarrow \tau
\end{align*}
\]

---

\( \text{meta-down} \) and \( \text{meta-up} \) correspond to \text{upML} and \text{downML} \([7]\), and are the compile-time version of \text{code} and \text{eval}.

\[
\begin{align*}
t &\Downarrow a \\
\text{meta-up}(t) &\Rightarrow a \\
\text{meta-down}^{-1}(t) &\Rightarrow \text{meta-up}(t)
\end{align*}
\]

The funcon \( \text{meta-down} \) triggers run-time evaluation at compile-time. At compile-time, \( \text{meta-down}(t) \) is replaced by \( t' \) if \( t \) compiles and evaluates to an AST \( a \) with \( a \Downarrow t' \).

\[
\begin{align*}
t &\Downarrow a \\
\text{meta-down}(t) &\Rightarrow t' \\
\text{meta-down}(t) &\Rightarrow t'
\end{align*}
\]

---

\( \text{meta-down} \) and \( \text{meta-up} \) correspond to \text{upML} and \text{downML} \([7]\), and are the compile-time version of \text{code} and \text{eval}.

\[
\begin{align*}
t &\Downarrow a \\
\text{meta-up}(t) &\Rightarrow a \\
\text{meta-down}(t) &\Rightarrow t'
\end{align*}
\]

The first argument is compiled and evaluated to an identifier \( i \). The second argument is compiled and evaluated to a value \( v \). The binding \( i \mapsto v \) is active in the compilation of the third argument \( t_3 \) to \( t'_3 \), which replaces \( \text{meta-let}(t_1, t_2, t_3) \) at compile-time. In Rule (24), we assume that bindings are propagated using the semantic entity \text{env} (environment),
495 \text{give(code(bound("x")), scope(bind("x", 7), eval(given)))}
496 \rightarrow \text{give(ast(tag(bound), astv(identifiers, "x")), scope(bind("x", 7), eval(given)))}
497 \rightarrow \text{give(astv(tag(bound), astv(identifiers, "x")), scope(bind("x", 7), eval(given)))}
498 \rightarrow \text{give(astv(tag(bound), astv(identifiers, "x")), scope(bind("x", 7), eval(astv(tag(bound), astv(identifiers, "x")))))}
499 \rightarrow \text{give(astv(tag(bound), astv(identifiers, "x")), scope(bind("x", 7), bound("x")))}
500 \rightarrow 7

Figure 3. An example of run-time evaluation of a funcon term with meta-programming.

\begin{align*}
e \in \text{exprs} & \coloneqq \ldots \\
| \text{eval } e & \quad \text{eval} \\
| \text{lift } e & \quad \text{lift} \\
| \text{let}_1 \ x = e_1 \ \text{in} \ e_2 & \quad \text{letd} \\
| \downarrow e & \quad \text{downML} \\
| \uparrow e & \quad \text{upML} \\
| \text{promote } e & \quad \text{promote} \\
| \text{astVar}(x) & \quad \text{ast-var} \\
| \text{astPlus}(e_1, e_2) & \quad \text{ast-plus}
\end{align*}

Figure 4. The extended abstract syntax of $\lambda_v$ expressions.

holding mappings between identifiers and values (as in [18]). We refer the reader to [9, 16] for the precise details of using environments in I-MSOS rules.

The funcons \text{meta-down}, \text{meta-up}, and \text{meta-let} have no run-time semantics; they are removed at compile-time.

4 Translating AST Constructors

In the previous section we have defined a funcon for most of the HGMP constructs of Berger, Tratt and Urban [7]. In this section we show that the funcons for HGMP are sufficiently powerful to apply the HGMP recipe to $\lambda_v$.

The main challenge of extending $\lambda_v$ is specifying the semantics of AST constructors. As the HGMP recipe reflects, HGMP languages often have an AST constructor for each construct of the language. This potentially causes a large amount of duplication in a formal definition of the semantics of the language (as well as in the syntax). We demonstrate that we can avoid this duplication in a component-based semantics given by (translation) functions in an algebra.

Figures 4 and 5 extend Figures 1 and 2 respectively. As examples of AST constructors, we have added $\text{ast-var}$ and $\text{ast-plus}$. Possible definitions of $\text{ast-var}$ and $\text{ast-plus}$ are:

\begin{align*}
\text{ast-var}(x) & = \text{ast(tag(current-value),} \\
& \quad \text{tag(bound), astv(identifiers.x))} \\
\text{ast-plus}(e_1, e_2) & = \text{ast(tag(integer-add),} e_1, e_2)
\end{align*}

(Note that the constructed AST representations are of funcon terms, not $\lambda_v$ expressions.) These definitions mirror the semantics of $\text{var}$ and $\text{plus}$ given in Figure 2, and a change

in the semantics of $\text{var}$ would require a similar change to the semantics of $\text{ast-var}$. To avoid this, we reuse functions $\text{var}$ and $\text{plus}$ in the definitions of $\text{ast-var}$ and $\text{ast-plus}$ respectively, as shown in Figure 5. By reusing $\text{ast-var}$ and $\text{ast-plus}$, we take advantage of the operational equivalence between $\lambda_v$ expressions and the funcon terms they translate to (the equivalence follows by definition).

The AST constructors $\text{ast-var}$ and $\text{ast-plus}$ construct AST representations at compile-time, as we have used $\text{meta-up}$ and $\text{meta-down}$ in their translation. If AST constructors construct AST representations at run-time, their translation should use code and eval instead.

Figure 5 gives semantics to two HGMP constructs with no direct funcon equivalent: lift and promote — for lifting values to ASTs and higher-order meta-programming respectively. Their semantics are expressed in terms of existing funcons and the funcons for other HGMP constructs.

5 Computational Abstractions as ASTs

In this section we further demonstrate the advantages of AST representations and funcons for HGMP. Firstly, we give semantics to call-by-name evaluation in $\lambda_v$. Secondly, we give

\footnote{We have focused on two levels: the level of programs and the meta-level of meta-representations. However, the funcons for HGMP support higher-order meta-programming in which infinitely many meta-levels are possible.}
The expressions double (fib 7) computes the seventh Fibonacci number, regardless of the definition of double. This may be inefficient, if double does not ‘use’ its parameter. In general, in call-by-value semantics, every argument is evaluated exactly once.

With the meta-programming constructs of λv, the programmer can decide, however, to delay the evaluation of arguments. For example, a programmer can write double (fib 7). This is the first step towards transforming the parameter of double into a call-by-name parameter. To complete the transformation, occurrences of the parameter within the body of double are wrapped with eval, forcing the evaluation of the argument where it is used. In general, the arguments provided for such call-by-name parameters are evaluated zero or more times. For example, if double is defined as let double = λn.eval n + eval n, then the expression fib 7 is evaluated twice when double (↑{fib 7}) is compiled and evaluated.

**Call-by-need semantics** We introduce a new language construct for transforming n into a lazy parameter. As discussed in Section 2, arguments are assigned to newly allocated references. Here we take advantage. We introduce !x as an alternative to eval x. The semantics of !x is to find the AST held by the reference r bound to x and evaluating the expression represented by the AST (equivalent to eval x). As a side-effect, the AST representation of the evaluation result replaces the AST held by r. The syntax and semantics of this construct are specified in Figure 6. In our example, if double is defined as let double = λn.!n + !n, then the evaluation of fib 7 is shared between the occurrences of n.

The funcons for HGMP can also be used to specify call-by-need evaluation in the semantics underlying λv. This is achieved by replacing e₂ in appₜ of Figure 2 by meta-up(e₂) (or code(e₂)), similarly replacing e₁ in letₜ by meta-up(e₁) (or code(e₁)), and defining varₜ as varₜ(x) = shareₜ(x) (with shareₜ defined in Figure 6). We expect that it is also possible to use the funcons for HGMP to specify the semantics of lazy parameters in Scala [17] and of strictness annotations in Haskell datatype declarations [13].

**6 Conclusions and future work**

In this paper we have developed funcons for building ASTs representing funcon terms and funcons for HGMP that act on these meta-representations. We demonstrated the power of the funcons for HGMP by giving semantics to call-by-need evaluation by transforming computations into AST representations to delay evaluation. The AST representation of funcon terms can also be used as the meta-representation of program fragments in the component-based semantics of languages, if the semantics has a reusable translation function for each language construct.

**Future work** We have implemented the relations ⌈⌉, |, and ⇒ as part of a funcon term interpreter [4]. On top of the funcon term interpreter, we have developed an interpreter for λv with all extensions, available online [6]. Translations functions such as varₜ and plusₜ are implemented directly in Haskell and are easily reused to implement ast-varₜ and ast-plusₜ. A future direction is to enable reuse translation functions in a specification language such as CBS, the specification language developed by the PlAnCompS project [5].

With these tools, we can study the coverage of the funcons for HGMP by defining component-based semantics for real-world programming languages as well as academic languages. Interesting targets in this investigation are MetaOCaml [12] and the reflective languages Black and Pink [1, 3]. MetaOCaml’s meta-programming constructs are similar to the constructs discussed in this paper and have been used in various applications [3, 11, 24, 25]. A reflective language has an underlying interpreter that gives semantics to the language, and programs can modify the underlying interpreter, thus changing the behaviour of programs as they are evaluated. Reflective languages therefore provide a significant stress-test to the component-based approach of programming language development with meta-programming.

**Acknowledgments**

The author thanks Martin Berger for discussions on HGMP, Peter Mosses and Neil Sculthorpe for discussions on funcons and component-based semantics, and Martin Berger, Peter Mosses, Neil Sculthorpe, Claudia Chirita, Duncan Mitchell and anonymous reviewers for their comments and suggestions on earlier versions of this paper.

**References**


