View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Royal Holloway - Pure

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:

L. Thomas van Binsbergen. 2018. Funcons for HGMP: The Fundamental Constructs of Homogeneous Generative Meta-Programming. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 λ-calculus as the running example. Section 3 defines the funcons for HGMP. Section 4 adds HGMP constructs to the λ-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 λ_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

[1] http://plancomps.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{rrcll}
x \in & \textbf{vars} & ::= & \ldots & \\
b \in & \textbf{bools} & ::= & \ldots & \\
i \in & \textbf{ints} & ::= & \ldots & \\
e \in & \textbf{exprs} & ::= & x & \textit{var} \\
 & & | & b & \textit{bool} \\
 & & | & i & \textit{int} \\
 & & | & \lambda x.e & \textit{lam} \\
 & & | & e_1\ e_2 & \textit{app} \\
 & & | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \textit{let} \\
 & & | & \textbf{ite}\ e_1\ e_2\ e_3 & \textit{ite} \\
 & & | & \textbf{this} & \textit{this} \\
 & & | & e_1 + e_2 & \textit{plus} \\
 & & | & e_1 \leqslant e_2 & \textit{leq}
\end{array}
$$

Figure 1. The syntax of λ_v.

documentation [18]. In Sections 4 and 5, λ_v is extended with HGMP constructs, for which we use the funcons developed in Section 3.

Homomorphic translations The following definitions are derived from [14]. Given a set S of sorts, an S-sorted signature Σ is a set of operations $f : (s_1, \ldots, s_n) \to s_0$ with $s_i \in S$, for all $1 \leqslant i \leqslant n$. Given an S-sorted signature Σ, a Σ-algebra A assigns a (carrier) set A_s to each sort $s \in S$ and a function $f_A : (A_{s_1}, \ldots, A_{s_n}) \to A_{s_0}$ to each operation $f : (s_1, \ldots, s_n) \to s_0$ in Σ. A Σ-homomorphism $h : A \to B$ (where A and B are Σ-algebras) assigns a total function $h_s : A_s \to B_s$ to $s \in S$ such that for each operation $f : (s_1, \ldots, s_n) \to s_0$ in Σ it holds that:

$$h_{s_0}(f_A(a_1, \ldots, a_n)) = f_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \quad (1)$$

A Σ-algebra I is initial in a class of Σ-algebras if there is a unique Σ-homomorphism from I to each algebra in the class [14]. An initial algebra in a class of algebras represents syntax; the other algebras in the class are possible semantics. An initial algebra can be constructed for each signature [10].

Abstract syntax Figure 1 defines a signature Σ_λ over the sorts **vars**, **bools**, **ints**, and **exprs**. We assume operations (constants) for **ints**, **bools** — the literals of λ_v — and **vars**. The operations for **exprs** are specified together with concrete syntax forms. For example, expressions of the form $\lambda x.e$ are represented by the operation $\textit{lam} : (\textbf{vars}, \textbf{exprs}) \to \textbf{exprs}$. Formally, the abstract syntax of λ_v is the union of the carrier sets of some initial Σ_λ-algebra $\mathscr{A}$.

Semantics Figure 2 defines a Σ_λ-algebra $\mathscr{F}$ by assigning functions to the operations of **exprs**, taking the set of all funcon terms as the carrier set for each of the sorts. The beta release of funcons has a rich universe of values, including **identifiers**, **integers**, and **booleans**. We omit functions assigning **identifiers**, **integers** and **booleans** to the operations of **vars**, **ints**, and **bools** respectively. Auxiliary

$$
\begin{aligned}
&\textit{this}_{\mathscr{F}} = \textbf{bound}(\texttt{"this"}) \\
&\textit{var}_{\mathscr{F}}(x) = \textbf{current-value}(\textbf{bound}(x)) \\
&\textit{bool}_{\mathscr{F}}(b) = b \\
&\textit{int}_{\mathscr{F}}(i) = i \\
&\textit{lam}_{\mathscr{F}}(x, e) = \\
&\ \textbf{function}(\textbf{closure}(\textit{let}_{\mathscr{F}}(x, \textit{given}_1, \textit{let}(\texttt{"this"}, \textit{given}_2, e)))) \\
&\textit{app}_{\mathscr{F}}(e_1, e_2) = \textbf{give}(e_1, \textbf{apply}(\textbf{given}, \textbf{tuple}(e_2, \textbf{given}))) \\
&\textit{let}_{\mathscr{F}}(x, e_1, e_2) = \textit{let}(x, \textbf{alloc-init}(\textbf{values}, e_1), e_2) \\
&\textit{ite}_{\mathscr{F}}(e_1, e_2, e_3) = \textbf{if-true-else}(e_1, e_2, e_3) \\
&\textit{plus}_{\mathscr{F}}(e_1, e_2) = \textbf{integer-add}(e_1, e_2) \\
&\textit{leq}_{\mathscr{F}}(e_1, e_2) = \textbf{integer-is-less-or-equal}(e_1, e_2) \\
&\textit{let}(x, e_1, e_2) = \textbf{scope}(\textbf{bind}(x, e_1), e_2) \\
&\textit{given}_1 = \textbf{first}(\textbf{tuple-elements}(\textbf{given})) \\
&\textit{given}_2 = \textbf{second}(\textbf{tuple-elements}(\textbf{given}))
\end{aligned}
$$

Figure 2. The semantics of λ_v, given as translation functions.

functions *let*, $\textit{given}_1$ and $\textit{given}_2$ are added for convenience. The homomorphism $\textit{fct} : \mathscr{A} \to \mathscr{F}$ translates λ_v programs into funcon terms. We obtain *fct* indirectly by defining $\mathscr{F}$ — rather than defining *fct* directly — which becomes useful when we reuse the functions of $\mathscr{F}$ in Section 4.

Variables in λ_v are bound to a value or to a reference holding a value. The references are initially redundant as λ_v does not have mutable variables. In Section 5, however, we extend the language and use the references to achieve sharing. Funcon **current-value** dereferences when its argument evaluates to a reference, otherwise the value of the argument is returned itself.

The first argument of *app* evaluates to a function[2], which is subsequently applied to a tuple. The first tuple element is e_2. The second tuple element is the function itself, thus enabling recursion. The combination of **give** and **given** specifies that e_1 evaluates once. The **give** funcon evaluates its first argument to a value which replaces occurrences of **given** within the second argument, unless these appear within the second argument of another occurrence of **give**. For example, the following funcon term evaluates to 5:

$$\textbf{give}(2, \textbf{integer-add}(\textbf{given}, \textbf{give}(\textbf{integer-add}(1, \textbf{given}), \textbf{given})))$$

Funcon **apply** evaluates its first argument to a function, its second argument to an arbitrary value v, and then **give**s v to the body of the function, i.e. for all terms b and values v, $\textbf{apply}(\textbf{function}(\textbf{abstraction}(b)), v)$ is equivalent to $\textbf{give}(v, b)$.

The function returned by a lambda-expression $\lambda x.e$ is statically scoped by computing a **closure** (rather than using

[2] We assume programs are well-typed.

substitution). The funcon **closure** computes an **abstraction** which restores the bindings that were active at the time it is computed. When the function is applied, the **identifier** x is bound to a reference holding the first element of the given tuple and binds **identifier** "this" to the second element of the given tuple. Thus, **this** can be used by programmers to make recursive calls, refering to the 'nearest' enclosing lambda (see $app_{\mathscr{F}}$).

3 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 λ_v. 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 $f(t_1, \ldots, t_n)$ is a value if and only f is in some subset C_V of C. A constructor in C_V is referred to as a value constructor, a constructor in $C_F = C \setminus C_V$ 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 $f(t_1, \ldots, t_n)$ is a type if f is in some subset C_T of C_V. A binary relation $\_ : \_$ between values and types expresses that a value v is of type t when $v : t$. We further assume a function $ty : C_V \rightarrow C_T$ that assigns a type to value v such that $v : ty(v)$. We make no assumptions about subtyping, i.e, $v : \tau \;\not\Rightarrow\; \tau = ty(v)$.

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 $\_ \rightarrow \_$. Finite evaluations are captured by the 'iterative closure' $\_ \dashrightarrow \_$ of $\_ \rightarrow \_$, expressing that c evaluates to value v when $c \dashrightarrow v$. The iterative closure is defined as:

[3]Examples of semantic entities are stores (heaps) and environments, for modelling imperative storage and variable bindings respectively.

$$\frac{f \in C_V}{f(t_1, \ldots, t_n) \dashrightarrow f(t_1, \ldots, t_n)} \quad (2) \qquad \frac{c_1 \rightarrow c_2 \quad c_2 \dashrightarrow v_1}{c_1 \dashrightarrow v_1} \quad (3)$$

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 $tag\langle f \rangle$. ASTs are formalised by the following rules.

$$\frac{v : \tau}{\mathbf{astv}(\tau, v) : \mathbf{asts}} \quad (4) \qquad \frac{a_1 : \mathbf{asts} \ldots a_n : \mathbf{asts}}{\mathbf{astv}(tag\langle f \rangle, a_1, \ldots, a_n) : \mathbf{asts}} \quad (5)$$

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**, **meta-down**, and **meta-let**. The additional types are **asts** and **tags**. The additional value constructors are **astv** and $tag\langle f \rangle$, with $tag\langle f \rangle : \mathbf{tags}$, for each computation constructor f. Let C', C'_F, C'_V, and C'_T be the extensions of C, C_F, C_V, and C_T respectively, and let C'_V replaces C_V 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 $\Downarrow_{dl}$, captures the conversion of a meta-representation a into the term t it represents. Relation $\_ \Downarrow \_$ is defined for computations by the following rule:

$$\frac{a_1 \Downarrow t_1 \ldots a_n \Downarrow t_n}{\mathbf{astv}(tag\langle f \rangle, a_1, \ldots, a_n) \Downarrow f(t_1, \ldots, t_n)} \quad (6)$$

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

The following rule defines $\_ \Downarrow \_$ for values:

$$\frac{v' = coerce(v, \tau)}{\mathbf{astv}(\tau, v) \Downarrow v'} \quad (7)$$

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 $v = coerce(v, \tau)$ for all v and τ.

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

astv(**booleans**, **true**). The dynamic semantics of **ast** is defined by the following rules:

$$\frac{v : \tau}{\mathbf{ast}(\tau, v) \rightarrow \mathbf{astv}(\tau, v)} \tag{8}$$

$$\frac{a_1 : \mathbf{asts} \dots a_n : \mathbf{asts}}{\mathbf{ast}(tag\langle f\rangle, a_1, \dots, a_n) \rightarrow \mathbf{astv}(tag\langle f\rangle, a_1, \dots, a_n)} \tag{9}$$

$$\frac{t_k \rightarrow t'_k}{\mathbf{ast}(t_1, \dots, t_k, \dots, t_n) \rightarrow \mathbf{ast}(t_1, \dots, t'_k, \dots, t_n)} \tag{10}$$

Rule (10) is a congruence rule, performing a (small-)step on one of the arguments of **ast**. This rule can be repeatedly applied until all arguments are evaluated (and no further, because $f \rightarrow f'$ implies that f is a computation). Rules (8) and (9) are applicable if all arguments are values, which follows from the conditions involving the typing relation.

We define the relation $\_ \Uparrow \_$, introduced in [7] as $\Downarrow_{ul}$, which captures the conversion of terms into their AST representation.

$$\frac{f \notin \{\mathbf{meta\text{-}down}, \mathbf{meta\text{-}up}\} \quad t_1 \Uparrow t'_1 \dots t_n \Uparrow t'_n}{f(t_1, \dots, t_n) \Uparrow \mathbf{ast}(tag\langle f\rangle, t'_1, \dots, t'_n)} \tag{11}$$

$$\frac{\tau = ty(v)}{v \Uparrow \mathbf{astv}(\tau, v)} \tag{12}$$

We give the cases $f = \mathbf{meta\text{-}down}$ and $f = \mathbf{meta\text{-}up}$ later, where we also show that the right-hand side of $\_ \Uparrow \_$ may be a partially evaluated AST.

Run-time HGMP The funcon **code** takes an arbitrary term t as argument and is dynamically replaced by the AST representation of t:

$$\frac{t \Uparrow a}{\mathbf{code}(t) \rightarrow a} \tag{13}$$

The funcon **eval** evaluates its argument to an AST a and is replaced by the term represented by a.

$$\frac{a \Downarrow t}{\mathbf{eval}(a) \rightarrow t} \tag{14} \qquad \frac{t \rightarrow t'}{\mathbf{eval}(t) \rightarrow \mathbf{eval}(t')} \tag{15}$$

As an example, consider the evaluation[4] in Figure 3.

The funcon **type-of** evaluates its argument to a value v and is replaced by the type $ty(v)$.

$$\frac{ty(v) = \tau}{\mathbf{type\text{-}of}(v) \rightarrow \tau} \tag{16} \qquad \frac{t \rightarrow t'}{\mathbf{type\text{-}of}(t) \rightarrow \mathbf{type\text{-}of}(t')} \tag{17}$$

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

[4]The rewrites of [18] have been omitted.

Compile-time HGMP The beta-release of funcons [18] does not include compile-time semantics. We proceed with the approach taken by Berger, Tratt and Urban [7] and define a relation $\_ \Rightarrow \_$, introduced as $\Downarrow_{ct}$ by the authors, which models a compilation phase. For funcons that do not involve compile-time meta-programming, the relation is defined as follows:

$$\frac{f \in C'_V}{f(t_1, \dots, t_n) \Rightarrow f(t_1, \dots, t_n)} \tag{18}$$

$$\frac{t_1 \Rightarrow t'_1 \dots t_n \Rightarrow t'_n \qquad f \notin \{\mathbf{meta\text{-}down}, \mathbf{meta\text{-}up}, \mathbf{meta\text{-}let}\} \quad f \notin C'_V}{f(t_1, \dots, t_n) \Rightarrow f(t'_1, \dots, t'_n)} \tag{19}$$

Rule (19) expresses that if f is not a funcon for compile-time meta-programming, nor a value constructor, then its subterms are compiled and possibly replaced. Rule (18) determines that values are not changed by compilation, even if it has computations as subterms.

The funcons **meta-up** and **meta-down** correspond to *upML* and *downML* [7], and are the compile-time version of **code** and **eval**.

$$\frac{t \Uparrow a}{\mathbf{meta\text{-}up}(t) \Rightarrow a} \tag{20} \qquad \frac{t_0 \Uparrow t_1 \quad t_1 \Uparrow t_2}{\mathbf{meta\text{-}up}(t_0) \Uparrow t_2} \tag{21}$$

The funcon **meta-down** triggers run-time evaluation at compile-time. At compile-time, $\mathbf{meta\text{-}down}(t_0)$ is replaced by t_2 if t compiles and evaluates to an AST a with $a \Downarrow t_2$.

$$\frac{t_0 \Rightarrow t_1 \quad t_1 \dashrightarrow a \quad a \Downarrow t_2}{\mathbf{meta\text{-}down}(t_0) \Rightarrow t_2} \tag{22} \qquad \frac{t \Rightarrow t'}{\mathbf{meta\text{-}down}(t) \Uparrow t'} \tag{23}$$

Rule (23) shows that an occurrence of **meta-down** within an occurrence of **meta-up** is 'cancelled out', resulting in a partially evaluated AST. For example, consider the computation **meta-up**(**give**(3, **meta-down**(**bound**("x")))), which compiles to $t = \mathbf{ast}(tag\langle\mathbf{give}\rangle, \mathbf{astv}(\mathbf{naturals}, 3), \mathbf{bound}(\texttt{"x"}))$. If t occurs in a context in which "x" is bound to an AST, then t evaluates to an AST. In this example, the computation **eval**(**scope**(**bind**("x", **code**(**given**)), t)) evaluates to 3.

In this example, "x" is bound at run-time. To bind identifiers at compile-time, we introduce **meta-let**, corresponding to *letdownML* [7]. It makes (non-local) bindings available, at compile-time, to occurrences of **meta-down**:

$$\frac{\begin{array}{c}\mathbf{env}(\rho) \vdash t_1 \Rightarrow t'_1 \quad \mathbf{env}(\rho) \vdash t'_1 \dashrightarrow i \\ \mathbf{env}(\rho) \vdash t_2 \Rightarrow t'_2 \quad \mathbf{env}(\rho) \vdash t'_2 \dashrightarrow v \\ \mathbf{env}(\rho[i \mapsto v]) \vdash t_3 \Rightarrow t'_3\end{array}}{\mathbf{env}(\rho) \vdash \mathbf{meta\text{-}let}(t_1, t_2, t_3) \Rightarrow t'_3} \tag{24}$$

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 $\mathbf{meta\text{-}let}(t_1, t_2, t_3)$ at compile-time. In Rule (24), we assume that bindings are propagated using the semantic entity **env** (environment),

$$
\begin{aligned}
& \mathbf{give}(\mathbf{code}(\mathbf{bound}(\texttt{"x"})), \mathbf{scope}(\mathbf{bind}(\texttt{"x"}, 7), \mathbf{eval}(\mathbf{given})))\\
\to\ & \mathbf{give}(\mathbf{ast}(\mathit{tag}\langle \mathbf{bound}\rangle, \mathbf{astv}(\mathbf{identifiers}, \texttt{"x"})), \mathbf{scope}(\mathbf{bind}(\texttt{"x"}, 7), \mathbf{eval}(\mathbf{given})))\\
\to\ & \mathbf{give}(\mathbf{astv}(\mathit{tag}\langle \mathbf{bound}\rangle, \mathbf{astv}(\mathbf{identifiers}, \texttt{"x"})), \mathbf{scope}(\mathbf{bind}(\texttt{"x"}, 7), \mathbf{eval}(\mathbf{given})))\\
\to\ & \mathbf{give}(\mathbf{astv}(\mathit{tag}\langle \mathbf{bound}\rangle, \mathbf{astv}(\mathbf{identifiers}, \texttt{"x"})), \mathbf{scope}(\mathbf{bind}(\texttt{"x"}, 7), \mathbf{eval}(\mathbf{astv}(\mathit{tag}\langle \mathbf{bound}\rangle, \mathbf{astv}(\mathbf{identifiers}, \texttt{"x"})))))\\
\to\ & \mathbf{give}(\mathbf{astv}(\mathit{tag}\langle \mathbf{bound}\rangle, \mathbf{astv}(\mathbf{identifiers}, \texttt{"x"})), \mathbf{scope}(\mathbf{bind}(\texttt{"x"}, 7), \mathbf{bound}(\texttt{"x"})))\\
\to\ & 7
\end{aligned}
$$

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

$$
\begin{array}{llll}
e \in & \mathbf{exprs} & ::= & \dots \\
& & \mid & \mathbf{eval}\ e \qquad \mathit{eval} \\
& & \mid & \mathbf{lift}\ e \qquad \mathit{lift} \\
& & \mid & \mathbf{let}_{\downarrow}\ x = e_1\ \mathbf{in}\ e_2 \qquad \mathit{letd} \\
& & \mid & \downarrow\{e\} \qquad \mathit{downML} \\
& & \mid & \uparrow\{e\} \qquad \mathit{upML} \\
& & \mid & \mathbf{promote}\ e \qquad \mathit{promote} \\
& & \mid & \mathbf{ast}_{var}(x) \qquad \mathit{ast\text{-}var} \\
& & \mid & \mathbf{ast}_{plus}(e_1, e_2) \qquad \mathit{ast\text{-}plus}
\end{array}
$$

Figure 4. The extended abstract syntax of λ_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 **meta-down**, **meta-up**, and **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 λ_v.

The main challenge of extending λ_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 $\mathbf{ast}_{var}$ and $\mathbf{ast}_{plus}$. Possible definitions of $\mathit{ast\text{-}var}_{\mathscr{F}}$ and $\mathit{ast\text{-}plus}_{\mathscr{F}}$ are:

$$
\begin{aligned}
\mathit{ast\text{-}var}_{\mathscr{F}}(x) &= \mathbf{ast}(\mathit{tag}\langle \mathbf{current\text{-}value}\rangle,\\
&\qquad \mathbf{ast}(\mathit{tag}\langle \mathbf{bound}\rangle, \mathbf{astv}(\mathbf{identifiers}, x)))\\
\mathit{ast\text{-}plus}_{\mathscr{F}}(e_1, e_2) &= \mathbf{ast}(\mathit{tag}\langle \mathbf{integer\text{-}add}\rangle, e_1, e_2)
\end{aligned}
$$

(Note that the constructed AST representations are of funcon terms, not λ_v expressions.) These definitions mirror the semantics of *var* and *plus* given in Figure 2, and a change in the semantics of *var* would require a similar change to the semantics of *ast-var*. To avoid this, we reuse functions $\mathit{var}_{\mathscr{F}}$ and $\mathit{plus}_{\mathscr{F}}$ in the definitions of $\mathit{ast\text{-}var}_{\mathscr{F}}$ and $\mathit{ast\text{-}plus}_{\mathscr{F}}$ respectively, as shown in Figure 5. By reusing $\mathit{ast\text{-}var}_{\mathscr{F}}$ and $\mathit{ast\text{-}plus}_{\mathscr{F}}$, we take advantage of the operational equivalence between λ_v expressions and the funcon terms they translate to (the equivalence follows by definition).

$$
\begin{aligned}
&\mathit{eval}_{\mathscr{F}}(e) = \mathbf{eval}(e)\\
&\mathit{lift}_{\mathscr{F}}(e) = \mathbf{give}(e, \mathit{lift}(\mathbf{given}))\\
&\mathit{letd}_{\mathscr{F}}(x, e_1, e_2) = \mathbf{meta\text{-}let}(x, e_1, e_2)\\
&\mathit{downML}_{\mathscr{F}}(e) = \mathbf{meta\text{-}down}(e)\\
&\mathit{upML}_{\mathscr{F}}(e) = \mathbf{meta\text{-}up}(e)\\
&\mathit{promote}_{\mathscr{F}}(e) = \mathbf{ast}(\mathbf{asts}, e)\\
&\mathit{ast\text{-}var}_{\mathscr{F}}(x) = \mathbf{meta\text{-}up}(\mathit{var}_{\mathscr{F}}(x))\\
&\mathit{ast\text{-}plus}_{\mathscr{F}}(e_1, e_2) =\\
&\quad \mathbf{meta\text{-}up}(\mathit{plus}_{\mathscr{F}}(\mathbf{meta\text{-}down}(e_1), \mathbf{meta\text{-}down}(e_2)))\\
&\mathit{lift}(e) = \mathbf{ast}(\mathbf{type\text{-}of}(e), e)
\end{aligned}
$$

Figure 5. Translation functions for the extended **exprs**.

The AST constructors $\mathbf{ast}_{var}$ and $\mathbf{ast}_{plus}$ construct AST representations at compile-time, as we have used **meta-up** and **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[5] 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 λ_v. Secondly, we give

[5]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.

$$e \in \textbf{exprs} ::= \ \ldots \ \mid\ !x \quad \textit{share}$$
$$\textit{share}_{\mathscr{F}}(x) = \textbf{give}(\textbf{eval}(\textbf{current-value}(\textbf{bound}(x))),\ \textbf{seq}(\textbf{assign}(\textbf{bound}(x), \textit{lift}(\textbf{given})), \textbf{given}))$$

Figure 6. A sharing construct based on funcons for HGMP.

semantics for call-by-need (lazy) evaluation as well, combining funcons for mutable references and HGMP, following the principle that 'call-by-need is call-by-name with sharing' [2]. Specifically, we show how meta-programming constructs make it possible for programmers to determine the evaluation strategy of each parameter. Funcons have not earlier been used to give semantics to call-by-need evaluation.

Call-by-name semantics Consider the definition of *fib* in the following λ_v fragment:

$$\textbf{let}\ \textit{fib} = \lambda n.\textbf{ite}\ (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))$$

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 λ_v, the programmer can decide, however, to delay the evaluation of arguments. For example, a programmer can write *double* ($\uparrow\{\textit{fib}\ 7\}$). 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* = $\lambda n.$**eval** n + **eval** n, then the expression *fib* 7 is evaluated twice when *double* ($\uparrow\{\textit{fib}\ 7\}$) 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* = $\lambda n.!n + !n$, 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 λ_v. This is achieved by replacing e_2 in $\textit{app}_{\mathscr{F}}$ of Figure 2 by **meta-up**(e_2) (or **code**(e_2)), similarly replacing e_1 in $\textit{let}_{\mathscr{F}}$ by **meta-up**(e_1) (or **code**(e_1)), and defining $\textit{var}_{\mathscr{F}}$ as $\textit{var}_{\mathscr{F}}(x) = \textit{share}_{\mathscr{F}}(x)$ (with $\textit{share}_{\mathscr{F}}$ 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 $\Uparrow$, $\Downarrow$, and $\Rightarrow$ as part of a funcon term interpreter [4]. On top of the funcon term interpreter, we have developed an interpreter for λ_v with all extensions, available online [6]. Translations functions such as $\textit{var}_{\mathscr{F}}$ and $\textit{plus}_{\mathscr{F}}$ are implemented directly in Haskell and are easily reused to implement $\textit{ast-var}_{\mathscr{F}}$ and $\textit{ast-plus}_{\mathscr{F}}$. A future direction is to enable reusing 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

[1] Nada Amin and Tiark Rompf. 2017. Collapsing Towers of Interpreters. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 52 (2017), 33 pages.

[2] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-need Lambda Calculus. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. 233–246.

[3] Kenichi Asai. 2014. Compiling a Reflective Language Using MetaOCaml. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. 113–122.

[4] Anonymous Author(s). 2016. Funcons Interpreter and Repository. https://hackage.haskell.org/package/funcons-tools. (2016). [Online, accessed 28-June-2018].

[5] Anonymous Author(s). 2016. Tool Support for Component-based Semantics. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM, 8–11.

[6] Anonymous Author(s). 2017. Interpreter for λ_v and extensions. https://hackage.haskell.org/package/funcons-lambda-cbv-mp. (2017). [Online, accessed 28-June-2018].

[7] Martin Berger, Laurence Tratt, and Christian Urban. 2017. Modelling homogeneous generative meta-programming. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP 2017)*. 5:1–5:23.

[8] Martin Berger, Laurence Tratt, and Andrei Voronkov. 2010. Program Logics for Homogeneous Meta-programming. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 64–81.

[9] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. 2015. Reusable Components of Semantic Specifications. In *Transactions on Aspect-Oriented Software Development XII*. 132–179.

[10] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. 1977. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM* 24, 1 (1977), 68–95.

[11] Jun Inoue, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Staging Beyond Terms: Prospects and Challenges. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. 103–108.

[12] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*. 86–102.

[13] Simon Marlow. 2010. Haskell 2010 Language Report. https://www.haskell.org/onlinereport/haskell2010/. (2010). [Online, accessed 08-June-2018].

[14] Peter Mosses. 1990. Denotational Semantics. In *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*, Jan van Leeuwen (Ed.). 577–631.

[15] Peter D. Mosses. 2004. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 195–228.

[16] Peter D. Mosses and Mark J. New. 2009. Implicit Propagation in Structural Operational Semantics. *Electronic Notes in Theoretical Computer Science* 229, 4 (2009), 49–66.

[17] Martin Odersky. 2018. Scala Language Specification. https://scala-lang.org/files/archive/spec/2.13/. (2018). [Online, accessed 08-June-2018].

[18] PLanCompS project. 2018. GitHub - Funcons-Beta. https://plancomps.github.io/CBS-beta/Funcons-beta. (2018). [Online, accessed 06-June-2018].

[19] Gordon D. Plotkin. 2004. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 17–139.

[20] Neil Sculthorpe, Paolo Torrini, and Peter D. Mosses. 2015. A Modular Structural Operational Semantics for Delimited Continuations. In *Proceedings of the Workshop on Continuations*. 63–80.

[21] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. *ACM SIGPLAN Notices* 37, 12 (2002), 60–75.

[22] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211 – 242.

[23] Laurence Tratt. 2005. Compile-time Meta-programming in a Dynamically Typed OO Language. In *Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05)*. ACM, 49–63.

[24] Takahisa Watanabe and Yukiyoshi Kameyama. 2018. Program generation for ML modules (short paper). In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. 60–66.

[25] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially static data as free extension of algebras. In *Proceedings of the 23rd ACM SIGPLAN International Conference on Functional Programming (ICFP 2018)*. ACM.