Funcons for HGMP
The Fundamental Constructs of Homogeneous Generative Meta-Programming (Short Paper)

L. Thomas van Binsbergen
Royal Holloway, University of London

6 November, 2018
Reusable Components of Semantic Specifications

Martin Churchill¹, Peter D. Mosses², Neil Sculthorpe², and Paolo Torrini²

¹ Google, Inc., London, UK
² PLANCOMPS Project, Swansea University, Swansea, UK
p.d.mosses@swansea.ac.uk
http://www.plancomps.org

- Identifies fundamental constructs in programming (paradigm-agnostic)
- Each funcon is formally defined via MSOS (Mosses, Plotkin)
- An open-ended library of (fixed) funcons makes Funcons
- Object language programs are translated to Funcons
- Unified meta-language called Component-Based Semantics (CBS)
Modelling Homogeneous Generative Meta-Programming*

Martin Berger¹, Laurence Tratt², and Christian Urban³

1 University of Sussex, Brighton, United Kingdom
2 King’s College London, United Kingdom
3 King’s College London, United Kingdom

HGMP: programs manipulate meta-representations of program fragments as data and choose when and where to evaluate

- Formalisation of HGMP through a λ-calculus
- A semi-mechanical HGMPification ‘recipe’ applicable to calculi
Research questions

*Can we apply the HGMPification recipe to Funcons?*

*What is the coverage of the added constructs?*

*Can we use them to describe real-world and academic languages?*

In this paper we demonstrate

The HGMPification of *Funcons* by introducing funcons for HGMP

The potential of combining existing funcons with funcons for HGMP

A possible semantics of AST constructors in an example language ($\lambda_v$)
1 Introduction to funcons and funcon translations through $\lambda_v$
2 HGMPification of FUNCONS and $\lambda_v$
3 An example of combining funcons for HGMP with existing funcons
The PLanCompS project

- The PLanCompS project has identified over a hundred funcons
  - Procedural: procedures, references, scoping, iteration
  - Functional: functions, bindings, datatypes, patterns
  - Abnormal control: exceptions, delimited continuations

- A beta-version is available: https://plancomps.github.io/CBS-beta/
Example language

\[
\begin{aligned}
x & \in \text{vars} & ::= & \ldots \\
b & \in \text{bools} & ::= & \ldots \\
i & \in \text{ints} & ::= & \ldots \\
e & \in \text{exprs} & ::= & x \\
\quad & & \mid & b \\
\quad & & \mid & i \\
\quad & & \mid & \lambda x. e \\
\quad & & \mid & e_1 \ e_2 \\
\quad & & \mid & \text{let } x = e_1 \ \text{in } e_2 \\
\quad & & \mid & \text{ite } e_1 \ e_2 \ e_3 \\
\quad & & \mid & \text{this} \\
\quad & & \mid & e_1 + e_2 \\
\quad & & \mid & e_1 \leq e_2
\end{aligned}
\]
\[
\begin{align*}
\text{exprs}[b] &= \text{bools}[b] \\
\text{exprs}[i] &= \text{ints}[i] \\
\text{exprs}[x] &= \text{current-value} (\text{bound}(\text{vars}[x])) \\
\text{exprs}[\text{let } x = e_1 \text{ in } e_2] &= \text{let}_{\text{var}} (\text{vars}[x], \text{exprs}[e_1], \text{exprs}[e_2])
\end{align*}
\]

\[
\begin{align*}
\text{let}_{\text{var}} (x, e_1, e_2) &= \text{let} (x, \text{alloc-init} (\text{values}, e_1), e_2) \\
\text{let} (x, e_1, e_2) &= \text{scope} (\text{bind} (x, e_1), e_2)
\end{align*}
\]
\[
\text{exprs}[^b] = \text{boots}[^b] \\
\text{exprs}[^i] = \text{ints}[^i] \\
\text{exprs}[^x] = \text{current-value} (\text{bound}(\text{vars}[^x])) \\
\text{exprs}[\text{let } x = e_1 \text{ in } e_2] = \text{let}_v a r (\text{vars}[^x], \text{exprs}[^e_1], \text{exprs}[^e_2]) \\
\]

\[
\text{let}_v a r (x, e_1, e_2) = \text{let} (x, \text{alloc-init} (\text{values}, e_1), e_2) \\
\text{let} (x, e_1, e_2) = \text{scope} (\text{bind} (x, e_1), e_2) \\
\]

Example

\[
\text{exprs}[\text{let } y = 1 \text{ in } y] = \text{scope} (\text{bind} ("y", \text{alloc-init} (\text{values}, 1)), "y") \\
\]
\[
\begin{align*}
\text{exprs}[\text{this}] &= \text{bound}"this""
\\
\text{exprs}[\lambda x. e] &= \text{function}(\text{closure}(\text{let}_\text{var}(\text{vars}[x], \text{given}_1, \text{let}"this", \text{given}_2, \text{exprs}[e])))
\\
\text{exprs}[e_1 e_2] &= \text{give}(\text{exprs}[e_1], \text{apply}(\text{given}, \text{tuple}(\text{exprs}[e_2], \text{given})))
\\
\text{given}_1 &= \text{first}(\text{tuple-elements}(\text{given}))
\\
\text{given}_2 &= \text{second}(\text{tuple-elements}(\text{given}))
\end{align*}
\]
\[
\text{exprs}[\text{this}] = \text{bound}(\text{"this"})
\]
\[
\text{exprs}[\lambda x. e] = \text{function(closure(let_var(vars[x], given_1, let(\text{"this"}, given_2, exprs[e]))))}
\]
\[
\text{exprs}[e_1 e_2] = \text{give(exprs[e_1], apply(given, tuple(exprs[e_2], given)))}
\]

\[
given_1 = \text{first(tuple-elements(given))}
\]
\[
given_2 = \text{second(tuple-elements(given))}
\]

**Example**
\[
\text{exprs}[(\lambda y. y) 1] = \text{give(function(closure(abs)), apply(given, tuple(1, given)))}
\]
\[
abs = \text{scope(bind("y", alloc-init(values, given_1)), scope(bind("this", given_2), body))}
\]
\[
body = \text{current-value(bound("y"))}
\]
exprs[ite $e_1$ $e_2$ $e_3$] = if-true-else($exprs[e_1], exprs[e_2], exprs[e_3]$)

exprs[$e_1 + e_2$] = integer-add($exprs[e_1], exprs[e_2]$)

exprs[$e_1 \leq e_2$] = integer-is-less-or-equal($exprs[e_1], exprs[e_2]$)
Overview

1. Introduction to funcons and funcon translations through $\lambda_V$
2. HGMPification of FUNCONS and $\lambda_V$
3. An example of combining funcons for HGMP with existing funcons
i) Add meta-representations (ASTs), with ↓ and ↑ modelling conversion

ii) Introduce a compilation phase, modelled by ⇒

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs
i) Add meta-representations (ASTs), with \( \downarrow \) and \( \uparrow \) modelling conversion
   New types: tags, asts

ii) Introduce a compilation phase, modelled by \( \Rightarrow \)

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs
i) Add meta-representations (ASTs), with ↓ and ↑ modelling conversion
   New types: tags, asts
   New funcons: ast, astv

ii) Introduce a compilation phase, modelled by ⇒

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs
i) Add meta-representations (ASTs), with ↓ and ↑ modelling conversion
   New types: tags, asts
   New funcons: ast, astv

ii) Introduce a compilation phase, modelled by ⇒
   (Run-time semantics of FUNCONS is modelled by →)

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs
i) Add meta-representations (ASTs), with ↓ and ↑ modelling conversion
   New types: tags, asts
   New funcons: ast, astv

ii) Introduce a compilation phase, modelled by ⇒
    (Run-time semantics of FUNCONS is modelled by →)

iii) Add compile-time HGMP constructs
    New funcons: meta-up, meta-down, meta-let

iv) Add run-time HGMP constructs
HGMPification

i) Add meta-representations (ASTs), with ↓ and ↑ modelling conversion
   New types: tags, asts
   New funcons: ast, astv

ii) Introduce a compilation phase, modelled by ⇒
    (Run-time semantics of FUNCONS is modelled by →→)

iii) Add compile-time HGMP constructs
    New funcons: meta-up, meta-down, meta-let

iv) Add run-time HGMP constructs
    New funcons: eval
Let \texttt{tags} be the type of funcon names (in our examples, \texttt{tags} are \texttt{strings})

- New value constructor \texttt{astv}(T, V_1, \ldots, V_k) for building ASTs, with
  - If \( T \) a type, then \( k = 1 \) and \( V_1 \) some value with \( V : T \)
  - If \( T \) a tag, then \( V_1, \ldots, V_k \) are \texttt{asts}

\[
\begin{align*}
T : \text{types} & \quad \Rightarrow \\
\text{astv}(T, V) \Downarrow V \\
T : \text{tags} \quad V_1 \Downarrow X_1 \ldots V_k \Downarrow X_k & \quad \Rightarrow \\
\text{astv}(T, V_1, \ldots, V_k) \Downarrow \text{funcon}_T(X_1, \ldots, X_k)
\end{align*}
\]
Dynamic semantics of meta-representations

- New function $\text{ast}(X_0, X_1, \ldots, X_k)$, with
  - $X_0$ (evaluates to) a tag or a type
  - $X_1, \ldots, X_k$ (evaluate to) a single value or zero or more $\text{asts}$

\[
\frac{T : \text{types} \quad V : T}{\ast(T, V) \rightarrow \ast v(T, V)}
\]

\[
\frac{T : \text{tags} \quad V_1 : \text{asts} \ldots V_n : \text{asts}}{\ast(T, V_1, \ldots, V_n) \rightarrow \ast v(T, V_1, \ldots, V_n)}
\]

\[
X_i \rightarrow X_i'
\]

\[
\frac{\ast(X_0, \ldots, X_i, \ldots, X_k)}{\ast(X_0, \ldots, X_i', \ldots, X_k)}
\]
<table>
<thead>
<tr>
<th>Funcon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>meta-up(e₁)</code></td>
<td>Convert $e₁$ to its meta-representation</td>
</tr>
<tr>
<td><code>meta-down(e₁)</code></td>
<td>Compile and evaluate $e₁$, and splice the resulting AST at this location</td>
</tr>
<tr>
<td><code>meta-down(e₁)</code></td>
<td>Cancels out upwards conversion, potentially leaving a partially evaluated AST</td>
</tr>
<tr>
<td><code>meta-let(x, e₁, e₂)</code></td>
<td>Bind $x$ to the value of $e₁$, making it available to $e₂$</td>
</tr>
<tr>
<td><code>eval(e₁)</code></td>
<td>Evaluate $e₁$ to an AST $a$, and evaluate the term represented by $a$</td>
</tr>
</tbody>
</table>
Example (1)

```
 eval(scope(bind("x", meta-up(given)))
      , meta-up(give(3, meta-down(bound("x"))))))
```

compiles to:

```
 eval(scope(bind("x", ast("given")))
      , ast("give", astv(naturals, 3), bound("x"))))
```

which evaluates to 3
Example (2)

```
print(meta-let("x"
    , astv(integers, read)
    , meta-down(ast("integer-add", astv(integers, 1), bound("x"))))))
```

compiles to:

```
print(integer-add(1, 7))
```

if the user inputs 7 during compilation
i) Add meta-representations (ASTs), with \( \downarrow \) and \( \uparrow \) modelling conversion
   New types: tags, asts
   New funcons: ast, astv

ii) Introduce a compilation phase, modelled by \( \Rightarrow \)
   (Run-time semantics of FUNCONS is modelled by \( \rightarrow \))

iii) Add compile-time HGMP constructs
   New funcons: meta-up, meta-down, meta-let

iv) Add run-time HGMP constructs
   New funcons: eval
HGMPification of $\lambda_v$

i) Add meta-representations (ASTs)

ii) Add compile-time HGMP constructs

iii) Add run-time HGMP constructs
Adding HGMP Constructs

\[ e \in \text{exprs} ::= \ldots \]
\[ \mid \text{eval } e \]
\[ \mid \text{lift } e \]
\[ \mid \text{let}_\downarrow x = e_1 \text{ in } e_2 \]
\[ \mid \downarrow\{e\} \]
\[ \mid \uparrow\{e\} \]

\[ \text{exprs}[\text{eval } e] = \text{eval}(\text{exprs}[e]) \]
\[ \text{exprs}[\text{lift } e] = \text{ast}(\text{values}, \text{exprs}[e]) \]
\[ \text{exprs}[\text{let}_\downarrow x = e_1 \text{ in } e_2] = \text{meta-let}(\text{vars}[x], \text{exprs}[e_1], \text{exprs}[e_2]) \]
\[ \text{exprs}[\downarrow\{e\}] = \text{meta-down}(\text{exprs}[e]) \]
\[ \text{exprs}[\uparrow\{e\}] = \text{meta-up}(\text{exprs}[e]) \]
HGMPification of $\lambda_V$

i) Add meta-representations (ASTs)

ii) Add compile-time HGMP constructs

iii) Add run-time HGMP constructs
Recall translation of application:

\[
exprs[e_1 e_2] = \text{give}(exprs[e_1], \text{apply}(\text{given}, \text{tuple}(exprs[e_2], \text{given})))
\]

How do we translate the AST constructor for application?

\[
exprs[\text{ast}_{app}(e_1, e_2)] = \text{ast}(\text{"give"}, exprs[M], \text{ast}(\text{"apply"}, \ldots))
\]

We have duplicated the translation of application...
A funcon-translation \( \Psi \) is homomorphic if for each object language operator \( o \) we have an \( f_o \) such that:

\[
\Psi(o(M_1, \ldots, M_k)) = f_o(\Psi(M_1), \ldots, \Psi(M_k))
\]

We can write the translation of application as follows:

\[
\mathit{exprs}[e_1 \ e_2] = f_{\mathit{app}}(\mathit{exprs}[e_1], \mathit{exprs}[e_2])
\]

\[
f_{\mathit{app}}(M, N) = \mathit{give}(M, \mathit{apply}(\mathit{given}, \mathit{tuple}(N, \mathit{given})))
\]

and the translation of the AST constructor as follows:

\[
\mathit{exprs}[\mathit{ast}_{\mathit{app}}(e_1, e_2)] = \mathit{meta-up}(f_{\mathit{app}}(\mathit{meta-down}(\mathit{exprs}[e_1]), \mathit{meta-down}(\mathit{exprs}[e_2])))
\]
Overview

1. Introduction to funcons and funcon translations through $\lambda_V$
2. HGMPification of Funcons and $\lambda_V$
3. An example of combining funcons for HGMP with existing funcons
let fib = \( n \leq 2 \) \( 1 (\text{this}(n + (-2)) + \text{this}(n + (-1))) \)

\text{in} let double = \( n \) \( n + n \)

\text{in} double (fib 7)
Delayed Arguments

Call-by-Value

\[
\text{let } \text{fib} = \lambda n. \text{ite} (n \leq 2) 1 (\text{this}(n + (-2)) + \text{this}(n + (-1))) \\
\text{in let } \text{double} = \lambda n. 13 + 13 \\
\text{in double (fib 7)}
\]
Call-by-Value

\[
\text{let } fib = \lambda n. \text{ite}\ (n \leq 2) 1 (\text{this}(n + (-2)) + \text{this}(n + (-1))) \\
\text{in let } double = \lambda n. 13 + 13 \\
\text{in double } (\uparrow \{fib\ 7\})
\]
Call-by-Value

\[
\text{let } \textit{fib} = \lambda n. \text{ite } (n \leq 2) 1 (\textit{this}(n + (-2)) + \textit{this}(n + (-1))) \\
\text{in let } \textit{double} = \lambda n. n + n \\
\text{in } \textit{double} (\uparrow\{\textit{fib} 7\})
\]
Delayed Arguments

Call-by-Value

\[
\text{let } fib = \lambda n. \text{ite} \ (n \leq 2) \ 1 \ (\text{this}(n + (-2)) + \text{this}(n + (-1))) \\
\text{in let } double = \lambda n. \text{eval} \ n + \text{eval} \ n \\
\text{in } double \ (\uparrow\{\text{fib} \ 7\})
\]
Call-by-Name

let fib = \n. \ite (n \leq 2) 1 (\this (n + (-2)) + \this (n + (-1)))
  in let double = \n. eval n + eval n
    in double (↑{fib 7})
let fib = \( n \mapsto \) \( \text{ite} \ (n \leq 2) \ 1 \ (\text{this}(n + (-2)) + \text{this}(n + (-1))) \) 

in let double = \( n \mapsto !x + !x \) 

in double (↑\{fib 7\})
Delayed Arguments

Call-by-Name + Sharing

\[
\begin{align*}
\texttt{let } fib & = \lambda n. \texttt{ite} (n \leq 2) 1 (\texttt{this} (n + (-2)) + \texttt{this} (n + (-1))) \\
\texttt{in let } double & = \lambda n. !x + !x \\
\texttt{in double} & (\uparrow\{ fib 7 \})
\end{align*}
\]

Sharing Construct

\[
e \in \texttt{exprs} \ ::= \ldots \\
| !x
\]

\[
\texttt{exprs}[!x] = \texttt{give(} \texttt{eval(} \texttt{current-value(} \texttt{bound(} vars[x] \texttt{)} \texttt{)}) \texttt{)}
\]

\[
, \texttt{seq(} \texttt{assign(} \texttt{bound(} vars[x] \texttt{)}, \texttt{ast(} values, \texttt{given} \texttt{)})\texttt{,} \texttt{given})\texttt{)}
\]
Conclusions

• Applying HGMPification to Funcons is relatively straightforward (details in paper)
• Adding object language ASTs risk duplication; solvable in homomorphic translations
• Potential benefits:
  - Widen the scope of the funcon approach to include HGMP languages
  - New method to formalise behaviour of programs in HGMP languages
Future work

Component-Based Semantics (CBS)
- More funcons: concurrency, non-determinism
- Static semantics of funcons

Funcons for Meta-programming
- Extend $\lambda_v$ with pattern matching
- Integrate ideas of this paper into CBS
- Run-time version of **meta-up**, i.e. run-time quotation
- Extend OCaml Light semantics with MetaOCaml constructs
- Further case studies to investigate coverage of funcons for HGMP
Funcons for HGMP
The Fundamental Constructs of Homogeneous Generative Meta-Programming (Short Paper)

L. Thomas van Binsbergen
Royal Holloway, University of London

6 November, 2018