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 Churchill[1], Peter D. Mosses[2(✉)], Neil Sculthorpe[2], and Paolo Torrini[2]

[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 Berger[1], Laurence Tratt[2], and Christian Urban[3]

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 (λ_{V})

Overview

1. Introduction to funcons and funcon translations through λ_{V}
2. HGMPification of FUNCONS and λ_{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

$$
\begin{array}{rlll}
x \in & \textbf{vars} & ::= & \ldots \\
b \in & \textbf{bools} & ::= & \ldots \\
i \in & \textbf{ints} & ::= & \ldots \\
e \in & \textbf{exprs} & ::= & x \\
 & & \mid & b \\
 & & \mid & i \\
 & & \mid & \lambda x.e \\
 & & \mid & e_1\ e_2 \\
 & & \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
 & & \mid & \textbf{ite}\ e_1\ e_2\ e_3 \\
 & & \mid & \textbf{this} \\
 & & \mid & e_1 + e_2 \\
 & & \mid & e_1 \leqslant e_2
\end{array}
$$

Translation (1)

$$\begin{aligned}
\mathit{exprs}[\![b]\!] &= \mathit{bools}[\![b]\!] \\
\mathit{exprs}[\![i]\!] &= \mathit{ints}[\![i]\!] \\
\mathit{exprs}[\![x]\!] &= \textbf{current-value}(\textbf{bound}(\mathit{vars}[\![x]\!])) \\
\mathit{exprs}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] &= \mathit{let}_{\mathit{var}}(\mathit{vars}[\![x]\!], \mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!])
\end{aligned}$$

$$\begin{aligned}
\mathit{let}_{\mathit{var}}(x, e_1, e_2) &= \mathit{let}(x, \textbf{alloc-init}(\textbf{values}, e_1), e_2) \\
\mathit{let}(x, e_1, e_2) &= \textbf{scope}(\textbf{bind}(x, e_1), e_2)
\end{aligned}$$

Translation (1)

$$
\begin{aligned}
\mathit{exprs}[\![b]\!] &= \mathit{bools}[\![b]\!] \\
\mathit{exprs}[\![i]\!] &= \mathit{ints}[\![i]\!] \\
\mathit{exprs}[\![x]\!] &= \textbf{current-value}(\textbf{bound}(\mathit{vars}[\![x]\!])) \\
\mathit{exprs}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] &= \mathit{let}_{\mathit{var}}(\mathit{vars}[\![x]\!], \mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathit{let}_{\mathit{var}}(x, e_1, e_2) &= \mathit{let}(x, \textbf{alloc-init}(\textbf{values}, e_1), e_2) \\
\mathit{let}(x, e_1, e_2) &= \textbf{scope}(\textbf{bind}(x, e_1), e_2)
\end{aligned}
$$

Example

$$\mathit{exprs}[\![\textbf{let } y = 1 \textbf{ in } y]\!] = \textbf{scope}(\textbf{bind}(\texttt{"y"}, \textbf{alloc-init}(\textbf{values}, 1)), \texttt{"y"})$$

Translation (2)

$$exprs[\![\textbf{this}]\!] = \textbf{bound}(\texttt{"this"})$$
$$exprs[\![\lambda x.e]\!] = \textbf{function}(\textbf{closure}(let_{var}(vars[\![x]\!], given_1, let(\texttt{"this"}, given_2, exprs[\![e]\!]))))$$
$$exprs[\![e_1\ e_2]\!] = \textbf{give}(exprs[\![e_1]\!], \textbf{apply}(\textbf{given}, \textbf{tuple}(exprs[\![e_2]\!], \textbf{given})))$$

$$given_1 = \textbf{first}(\textbf{tuple-elements}(\textbf{given}))$$
$$given_2 = \textbf{second}(\textbf{tuple-elements}(\textbf{given}))$$

Translation (2)

$$\begin{aligned}
\mathit{exprs}[\![\textbf{this}]\!] &= \textbf{bound}(\texttt{"this"}) \\
\mathit{exprs}[\![\lambda x.e]\!] &= \textbf{function}(\textbf{closure}(\mathit{let}_{\mathit{var}}(\mathit{vars}[\![x]\!], \mathit{given}_1, \mathit{let}(\texttt{"this"}, \mathit{given}_2, \mathit{exprs}[\![e]\!])))) \\
\mathit{exprs}[\![e_1\ e_2]\!] &= \textbf{give}(\mathit{exprs}[\![e_1]\!], \textbf{apply}(\textbf{given}, \textbf{tuple}(\mathit{exprs}[\![e_2]\!], \textbf{given})))
\end{aligned}$$

$$\begin{aligned}
\mathit{given}_1 &= \textbf{first}(\textbf{tuple-elements}(\textbf{given})) \\
\mathit{given}_2 &= \textbf{second}(\textbf{tuple-elements}(\textbf{given}))
\end{aligned}$$

Example

$$\begin{aligned}
\mathit{exprs}[\![(\lambda y.y)\ 1]\!] &= \textbf{give}(\textbf{function}(\textbf{closure}(\mathit{abs})), \textbf{apply}(\textbf{given}, \textbf{tuple}(1, \textbf{given}))) \\
\mathit{abs} &= \textbf{scope}(\textbf{bind}(\texttt{"y"}, \textbf{alloc-init}(\textbf{values}, \mathit{given}_1)) \\
&\qquad , \textbf{scope}(\textbf{bind}(\texttt{"this"}, \mathit{given}_2), \mathit{body})) \\
\mathit{body} &= \textbf{current-value}(\textbf{bound}(\texttt{"y"}))
\end{aligned}$$

Translation (3)

$$\mathit{exprs}[\![\textbf{ite}\ e_1\ e_2\ e_3]\!] = \textbf{if-true-else}(\mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!], \mathit{exprs}[\![e_3]\!])$$

$$\mathit{exprs}[\![e_1 + e_2]\!] = \textbf{integer-add}(\mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!])$$

$$\mathit{exprs}[\![e_1 \leqslant e_2]\!] = \textbf{integer-is-less-or-equal}(\mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!])$$

Overview

1. Introduction to funcons and funcon translations through λ_{V}
2. HGMPification of FUNCONS and λ_{V}
3. An example of combining funcons for HGMP with existing funcons

HGMPification

i) Add meta-representations (ASTs), with $\Downarrow$ and $\Uparrow$ modelling conversion

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

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs

HGMPification

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

HGMPification

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 $\Rightarrow$

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs

HGMPification

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 $\longrightarrow$)

iii) Add compile-time HGMP constructs

iv) Add run-time HGMP constructs

HGMPification

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 $\longrightarrow$)

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 $\Downarrow$ 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 $\longrightarrow$)

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

iv) Add run-time HGMP constructs
   New funcons: **eval**

Meta-representations (ASTs)

Let **tags** be the type of funcon names (in our examples, **tags** are **strings**)

- New value constructor **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 **asts**

$$\frac{T : \textbf{types}}{\textbf{astv}(T, V) \Downarrow V}$$

$$\frac{T : \textbf{tags} \qquad V_1 \Downarrow X_1 \ldots V_k \Downarrow X_k}{\textbf{astv}(T, V_1, \ldots, V_k) \Downarrow \textbf{funcon}_T(X_1, \ldots, X_k)}$$

Dynamic semantics of meta-representations

- New funcon **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 **asts**

$$\frac{T : \textbf{types} \quad V : T}{\textbf{ast}(T, V) \longrightarrow \textbf{astv}(T, V)}$$

$$\frac{T : \textbf{tags} \qquad V_1 : \textbf{asts} \ldots V_n : \textbf{asts}}{\textbf{ast}(T, V_1, \ldots, V_n) \longrightarrow \textbf{astv}(T, V_1, \ldots, V_n)}$$

$$\frac{X_i \longrightarrow X_i'}{\textbf{ast}(X_0, \ldots, X_i, \ldots, X_k) \longrightarrow \textbf{ast}(X_0, \ldots, X_i', \ldots, X_k)}$$

Funcons for HGMP - Summary

Funcon	Description
meta-up(e_1)	Convert e_1 to its meta-representation
meta-down(e_1) (outside **meta-up**)	Compile and evaluate e_1, and splice the resulting AST at this location
meta-down(e_1) (inside **meta-up**)	Cancels out upwards conversion, potentially leaving a partially evaluated AST
meta-let(x, e_1, e_2)	Bind x to the value of e_1, making it available to e_2
eval(e_1)	Evaluate e_1 to an AST a, and evaluate the term represented by a

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

HGMPification (completed)

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 $\longrightarrow$)

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 λ_{V}

i) Add meta-representations (ASTs)

ii) Add compile-time HGMP constructs

iii) Add run-time HGMP constructs

Adding HGMP Constructs

$$
\begin{array}{rcll}
e \in & \textbf{exprs} & ::= & \ldots \\
 & & | & \textbf{eval}\ e \\
 & & | & \textbf{lift}\ e \\
 & & | & \textbf{let}_{\downarrow}\ x = e_1\ \textbf{in}\ e_2 \\
 & & | & \downarrow\{e\} \\
 & & | & \uparrow\{e\}
\end{array}
$$

$$
\begin{aligned}
\mathit{exprs}[\![\textbf{eval}\ e]\!] &= \textbf{eval}(\mathit{exprs}[\![e]\!]) \\
\mathit{exprs}[\![\textbf{lift}\ e]\!] &= \textbf{ast}(\textbf{values}, \mathit{exprs}[\![e]\!]) \\
\mathit{exprs}[\![\textbf{let}_{\downarrow}\ x = e_1\ \textbf{in}\ e_2]\!] &= \textbf{meta-let}(\mathit{vars}[\![x]\!], \mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!]) \\
\mathit{exprs}[\![\downarrow\{e\}]\!] &= \textbf{meta-down}(\mathit{exprs}[\![e]\!]) \\
\mathit{exprs}[\![\uparrow\{e\}]\!] &= \textbf{meta-up}(\mathit{exprs}[\![e]\!])
\end{aligned}
$$

HGMPification of λ_{V}

i) Add meta-representations (ASTs)

ii) Add compile-time HGMP constructs

iii) Add run-time HGMP constructs

Adding AST Constructors

Recall translation of application:

$$\mathit{exprs}[\![e_1\ e_2]\!] = \textbf{give}(\mathit{exprs}[\![e_1]\!], \textbf{apply}(\textbf{given}, \textbf{tuple}(\mathit{exprs}[\![e_2]\!], \textbf{given})))$$

How do we translate the AST constructor for application?

$$\mathit{exprs}[\![\textbf{ast}_{app}(e_1, e_2)]\!] = \textbf{ast}(\texttt{"give"}, \mathit{exprs}[\![M]\!], \textbf{ast}(\texttt{"apply"}, \ldots))$$

We have duplicated the translation of application...

Homomorphic Translations

A funcon-translation Ψ 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:

$$\begin{aligned} \mathit{exprs}[\![e_1\ e_2]\!] &= f_{app}(\mathit{exprs}[\![e_1]\!], \mathit{exprs}[\![e_2]\!]) \\ f_{app}(M, N) &= \textbf{give}(M, \textbf{apply}(\textbf{given}, \textbf{tuple}(N, \textbf{given}))) \end{aligned}$$

and the translation of the AST constructor as follows:

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

Overview

1. Introduction to funcons and funcon translations through λ_{V}
2. HGMPification of FUNCONS and λ_{V}
3. An example of combining funcons for HGMP with existing funcons

Delayed Arguments

Call-by-Value

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ n + n\\ &\qquad\textbf{in } \mathit{double}\ (\mathit{fib}\ 7)\end{aligned}$$

Delayed Arguments

Call-by-Value

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ 13 + 13\\ &\qquad\textbf{in } \mathit{double}\ (\mathit{fib}\ 7)\end{aligned}$$

Delayed Arguments

Call-by-Value

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ 13 + 13\\ &\qquad\textbf{in } \mathit{double}\ (\uparrow\{\mathit{fib}\ 7\})\end{aligned}$$

Delayed Arguments

Call-by-Value

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ n + n\\ &\qquad\textbf{in } \mathit{double}\ (\uparrow\{\mathit{fib}\ 7\})\end{aligned}$$

Delayed Arguments

Call-by-Value

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ \textbf{eval } n + \textbf{eval } n\\ &\qquad\textbf{in } \mathit{double}\ (\uparrow\{\mathit{fib}\ 7\})\end{aligned}$$

Delayed Arguments

Call-by-Name

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ \textbf{eval } n + \textbf{eval } n\\ &\qquad\textbf{in } \mathit{double}\ (\uparrow\{\mathit{fib}\ 7\})\end{aligned}$$

Delayed Arguments

Call-by-Name + Sharing

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ !x + !x\\ &\qquad\textbf{in } \mathit{double}\ (\uparrow\{\mathit{fib}\ 7\})\end{aligned}$$

Delayed Arguments

Call-by-Name + Sharing

$$\begin{aligned}&\textbf{let } \mathit{fib} = \lambda n.\textbf{ite } (n \leqslant 2)\ 1\ (\textbf{this}(n + (-2)) + \textbf{this}(n + (-1)))\\ &\quad\textbf{in let } \mathit{double} = \lambda n.\ !x + !x\\ &\qquad\textbf{in } \mathit{double}\ (\uparrow\{\mathit{fib}\ 7\})\end{aligned}$$

Sharing Construct

$$\begin{aligned}e \in\ \textbf{exprs}\ &::=\ \ldots\\ &\mid\ !x\end{aligned}$$

$$\begin{aligned}\mathit{exprs}[\![!x]\!] = \textbf{give}(&\textbf{eval}(\textbf{current-value}(\textbf{bound}(\mathit{vars}[\![x]\!])))\\ &,\textbf{seq}(\textbf{assign}(\textbf{bound}(\mathit{vars}[\![x]\!]),\textbf{ast}(\textbf{values},\textbf{given})),\textbf{given}))\end{aligned}$$

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 λ_{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