# The Fundamental Constructs of Homogeneous Generative Meta-Programming

## *or* Funcons for HGMP

L. Thomas van Binsbergen

Royal Holloway, University of London

17 January, 2018

## 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 the $\lambda$-calculus
- A HGMPification 'recipe' applicable to formal specifications

# 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

## Research Questions

*Can we apply HGMPification to* FUNCONS*?*

*Does this simplify giving a (component-based) semantics for languages with meta-programming facilities?*

Section 1

# HGMP

## $\lambda$-calculus with HGMP

- $\lambda$ programs generate (abstract syntax rep.) of $\lambda$ fragments
- The generated fragments may be inserted into the program

## Running Example

$$
\begin{array}{l}
\textbf{let}_{ct}\ \mathit{gen} = \lambda n.\textbf{if}\ n \leqslant 0\ \textbf{then}\ \ \uparrow\{0\} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \textbf{else}\ \ \uparrow\{x + \downarrow\{\textbf{this}\ (n-1)\}\} \\
\textbf{in let}_{ct}\ \mathit{product} = \lambda n. \uparrow\{\lambda x. \downarrow\{\mathit{gen}\ n\}\} \\
\quad \textbf{in}\ \ \downarrow\{\mathit{product}\ 3\}\ 8
\end{array}
$$

compiles to $(\lambda x.x + x + x + 0)\ 8$ and evaluates to $24$

## Syntax

$$M ::= x \mid M\ N \mid \lambda x.M \mid c \mid M + N \mid \ldots$$

## Dynamic Semantics

$$\frac{M \Downarrow_\lambda \lambda x.M' \qquad N \Downarrow_\lambda N' \qquad M'[N'/x] \Downarrow_\lambda V}{MN \Downarrow_\lambda V}$$

$$\frac{M \Downarrow_\lambda I_1 \qquad N \Downarrow_\lambda I_2}{M + N \Downarrow_\lambda I_1 +_{\mathbb{Z}} I_2}$$

$$\ldots$$

## Syntax

$$M ::= x \mid M\ N \mid \lambda x.M \mid c \mid M + N \mid \ldots$$

## Static Semantics

$$\frac{M \Downarrow_{ct} M' \qquad N \Downarrow_{ct} N'}{MN \Downarrow_{ct} M'N'}$$

$$\frac{M \Downarrow_{ct} M'}{\lambda x.M \Downarrow_{ct} \lambda x.M'}$$

$$\ldots$$

## Abstract syntax trees: syntax

$$t \;::= var \mid app \mid lam \mid int \mid string \mid add \mid \ldots$$
$$M ::= \ldots \mid ast_t\ (M_1 \ldots M_k)\ \textbf{where}\ k = arity\ (t)$$

## Abstract syntax trees: semantics

$$\frac{M_1 \Downarrow_\lambda M_1' \qquad \ldots \qquad M_k \Downarrow_\lambda M_k'}{ast_t(M_1 \ldots M_k) \Downarrow_\lambda ast_t(M_1' \ldots M_k')}$$

## Abstract syntax trees: syntax

$$t \;::=\; \mathit{var} \mid \mathit{app} \mid \mathit{lam} \mid \mathit{int} \mid \mathit{string} \mid \mathit{add} \mid \ldots$$

$$M ::= \ldots \mid \mathit{ast}_t\ (M_1 \ldots M_k)\ \textbf{where}\ k = \mathit{arity}\ (t)$$

## Abstract syntax trees: semantics

$$\frac{M_1 \Downarrow_{ct} M_1' \qquad \ldots \qquad M_k \Downarrow_{ct} M_k'}{\mathit{ast}_t(M_1 \ldots M_k) \Downarrow_{ct} \mathit{ast}_t(M_1' \ldots M_k')}$$

## Examples

$$ast_{app}(ast_{lam}(\texttt{"x"}, ast_{var}(\texttt{"x"})), ast_{int}(3))$$

is a value

$$ast_{add}(ast_{var}(\texttt{"x"}), (\lambda x.ast_{int}(x))\ 2)$$

evaluates to $ast_{add}(ast_{var}(\texttt{"x"}), ast_{int}(2))$

## downML $\downarrow\{...\}$ (splicing)

$$\downarrow\{ast_{app}(ast_{lam}(\texttt{"x"}, ast_{var}(\texttt{"x"})), ast_{int}(3))\}$$

compiles to $(\lambda x.x)\ 3$ and evaluates to $3$

## downML syntax

$$M ::= \ldots \mid \downarrow\{M\}$$

## downML semantics

$$\frac{M \Downarrow_{ct} M' \qquad M' \Downarrow_{\lambda} A \qquad A \Downarrow_{dl} N}{\downarrow\{M\} \Downarrow_{ct} N}$$

$$\frac{M \Downarrow_{dl} M' \qquad N \Downarrow_{dl} N'}{ast_{app}(M, N) \Downarrow_{dl} M' N'}$$

$$\frac{M \Downarrow_{dl} \text{“}x\text{”} \qquad N \Downarrow_{dl} N'}{ast_{lam}(M, N) \Downarrow_{dl} \lambda x.N'}$$

$$\ldots$$

To write *meaningful* programs *easily* we need:

- A way to bind names to $\lambda$ terms at compile time
- Backquoting / quasi-quoting, for conveniently writing ASTs
- Recursion
- Conditional choice
- More operators

**Extension syntax**

$$
\begin{aligned}
M ::= \ldots \mid & \ \mathbf{let}_{ct}\ x = M\ \mathbf{in}\ N \mid \ \uparrow\{M\} \\
\mid & \ \mathbf{this} \mid \mathbf{if}\ M\ \mathbf{then}\ N\ \mathbf{else}\ N' \mid M \leqslant N \mid M - N \mid \ldots
\end{aligned}
$$

## upML semantics

$$\frac{M \Downarrow_{ul} M'}{\uparrow\{M\} \Downarrow_{ct} M'}$$

$$\frac{M \Downarrow_{ul} M' \qquad N \Downarrow_{ul} N'}{MN \Downarrow_{ul} ast_{app}(M', N')}$$

$$\frac{M \Downarrow_{ct} M'}{\downarrow\{M\} \Downarrow_{ul} M'}$$

...

## Example

$$\begin{array}{l} \textbf{let}_{ct}\ \mathit{gen} = \lambda n.\textbf{if}\ n \leqslant 0\ \textbf{then}\ \uparrow\{0\} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \uparrow\{x + \downarrow\{\textbf{this}\ (n-1)\}\} \\ \textbf{in let}_{ct}\ \mathit{product} = \lambda n. \uparrow\{\lambda x. \downarrow\{\mathit{gen}\ n\}\} \\ \quad \textbf{in}\ \downarrow\{\mathit{product}\ 3\}\ 8 \end{array}$$

compiles to $(\lambda x.x + x + x + 0)\ 8$ and evaluates to $24$

## Example

$$\begin{array}{l} \textbf{let}_{ct}\ gen = \lambda n.\textbf{if}\ n \leqslant 0\ \textbf{then}\ \uparrow\{0\} \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \uparrow\{x + \downarrow\{\textbf{this}\ (n-1)\}\} \\ \textbf{in let}_{ct}\ product = \lambda n. \uparrow\{\lambda x. \downarrow\{gen\ n\}\} \\ \quad \textbf{in}\ \downarrow\{product\ 3\}\ 8 \end{array}$$

compiles to $(\lambda x.x + x + x + 0)\ 8$ and evaluates to $24$

## Halfway compilation:

$$\begin{array}{l} \downarrow\{product\ 3\}\ 8 \\ \textbf{with}\ product = \lambda n.ast_{lam}(ast_{string}(\texttt{"x"}), gen\ n) \\ \textbf{and}\ \ gen = \lambda n.\textbf{if}\ n \leqslant 0\ \textbf{then}\ ast_{int}(0) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ ast_{add}(ast_{var}(\texttt{"x"}), \textbf{this}\ (n-1)) \end{array}$$

## Run-time HGMP

$$\begin{array}{l}
\textbf{let } \mathit{gen} = \lambda n.\textbf{if } n \leqslant 0 \textbf{ then } \uparrow\{0\} \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } \uparrow\{x + \downarrow\{\textbf{this } (n-1)\}\} \\
\textbf{in let } \mathit{product} = \lambda n. \uparrow\{\lambda x. \downarrow\{\mathit{gen}\ n\}\} \\
\quad \textbf{in } (\textbf{eval } (\mathit{product}\ 3))\ 8
\end{array}$$

## After compilation

$$\begin{array}{l}
\textbf{let } \mathit{gen} = \lambda n.\textbf{if } n \leqslant 0 \textbf{ then } \mathit{ast}_{\mathit{int}}(0) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } \mathit{ast}_{\mathit{add}}(\mathit{ast}_{\mathit{var}}(\texttt{"x"}), \textbf{this } (n-1)) \\
\textbf{in let } \mathit{product} = \lambda n.\mathit{ast}_{\mathit{lam}}(\mathit{ast}_{\mathit{string}}(\texttt{"x"}), \mathit{gen}\ n) \\
\quad \textbf{in } (\textbf{eval } (\mathit{product}\ 3))\ 8
\end{array}$$

## Eval syntax

$$M ::= \ldots \mid \textbf{eval}\,(M)$$
$$t \;\; ::= \textbf{eval}$$

## Eval semantics

$$\frac{M \Downarrow_{\lambda} A \qquad A \Downarrow_{dl} N \qquad N \Downarrow_{\lambda} V}{\textbf{eval}(M) \Downarrow_{\lambda} V} \qquad \frac{A \Downarrow_{dl} M}{ast_{\textbf{eval}}\,(A) \Downarrow_{dl} \textbf{eval}(M)}$$

$$\frac{M \Downarrow_{ct} N}{\textbf{eval}(M) \Downarrow_{ct} \textbf{eval}(N)} \qquad \frac{M \Downarrow_{ul} A}{\textbf{eval}(M) \Downarrow_{ul} ast_{\textbf{eval}}(A)}$$

# Section 2

## Funcons

- 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 to be published: `plancomps.org`
- A semantics is obtained by translation to Funcons

$$
\begin{aligned}
\mathit{fct}[\![\textbf{if } G \textbf{ then } M \textbf{ else } N]\!] &= \textbf{if-then-else}(\mathit{fct}[\![G]\!], \mathit{fct}[\![M]\!], \mathit{fct}[\![N]\!]) \\
\mathit{fct}[\![M + N]\!] &= \textbf{integer-add}(\mathit{fct}[\![M]\!], \mathit{fct}[\![N]\!]) \\
\mathit{fct}[\![M \leqslant N]\!] &= \textbf{is-less-or-equal}(\mathit{fct}[\![M]\!], \mathit{fct}[\![N]\!]) \\
\ldots &= \ldots
\end{aligned}
$$

- Potential benefits of FUNCONS:
  - Development and maintenance of formal specifications
  - Teach and compare programming constructs across paradigms

# Funcons

A Funcons program (funcon term) is either:

- A value, e.g. **true**, $1$, $\{1, 2, 3\}$, **abs**(...), $\{\texttt{"x"} \mapsto$ **abs**(...)$\}$
- A computation: a funcon-name applied to funcon terms, e.g.

$$\textbf{seq}\ (\textbf{assign}\ (\textbf{bound}\ (\texttt{"x"})\ ,\ \textbf{integer-add}\ (\textbf{assigned}\ (\textbf{bound}\ (\texttt{"x"})), 1))\ ,\ \textbf{print}\ (\textbf{assigned}\ (\textbf{bound}\ (\texttt{"x"}))))$$

Funcon terms are freely composed:

- Many funcons are *variadic*
- But composition must satisfy funcon signatures

- The semantics of a Funcon is defined via small-step MSOS.
- MSOS rules are modular wrt *auxiliary entities*, modelling context and effects, e.g. *environment*, *store*, *output*, *control*, etc.

**assigned** (**bound** ("x")) $\rightarrow$ **assigned** (**variable** ($\#1$)) $\rightarrow$ 7

under any **environment** binding "x" to **variable** ($\#1$)

for any **store** with value $7$ at location $\#1$.

$$\mathit{fct}[\![\textbf{let } x = M \textbf{ in } N]\!] \;=\; \textbf{scope}(\textbf{bind}(x, \mathit{fct}[\![M]\!]), \mathit{fct}[\![N]\!])$$

| **funcon** | **informal semantics** |
|---|---|
| **bind** $(X, Y)$ | yield the environment binding identifier $\overline{X}$ to $\overline{Y}$ |
| **scope** $(X, Y)$ | evaluate $Y$ extending the current **environment** with the bindings in environment $\overline{X}$ |

$$\textit{fct}[\![M\ N]\!] = \mathbf{apply}(\textit{fct}[\![M]\!], \textit{fct}[\![N]\!]) \tag{v1}$$

(v2)

(v3)

| funcon | informal semantics |
|---|---|
| **given** | yield the current **given-value** |
| **give** $(X, Y)$ | evaluate $Y$ with **given-value** $\overline{X}$ |
| **abs** $(X)$ | a value constructor wrapping a computation $X$ |
| **apply** $(X, Y)$ | unwrap abstraction $\overline{X}$ and give $\overline{Y}$ to it |

$$fct[\![M\ N]\!] = \textbf{apply}(fct[\![M]\!], fct[\![N]\!]) \quad \text{(v1)}$$

$$fct[\![M\ N]\!] = \textbf{apply}(fct[\![M]\!], (fct[\![N]\!], fct[\![M]\!])) \quad \text{(v2)}$$

$$\text{(v3)}$$

| **funcon** | **informal semantics** |
|---|---|
| **given** | yield the current **given-value** |
| **give** $(X, Y)$ | evaluate $Y$ with **given-value** $\overline{X}$ |
| **abs** $(X)$ | a value constructor wrapping a computation $X$ |
| **apply** $(X, Y)$ | unwrap abstraction $\overline{X}$ and give $\overline{Y}$ to it |

$$\mathit{fct}[\![M\ N]\!] = \mathbf{apply}(\mathit{fct}[\![M]\!], \mathit{fct}[\![N]\!]) \tag{v1}$$

$$\mathit{fct}[\![M\ N]\!] = \mathbf{apply}(\mathit{fct}[\![M]\!], (\mathit{fct}[\![N]\!], \mathit{fct}[\![M]\!])) \tag{v2}$$

$$\mathit{fct}[\![M\ N]\!] = \mathbf{give}(\mathit{fct}[\![M]\!], \mathbf{apply}(\mathbf{given}, (\mathit{fct}[\![N]\!], \mathbf{given}))) \tag{v3}$$

| **funcon** | **informal semantics** |
|---|---|
| **given** | yield the current **given-value** |
| **give** $(X, Y)$ | evaluate $Y$ with **given-value** $\overline{X}$ |
| **abs** $(X)$ | a value constructor wrapping a computation $X$ |
| **apply** $(X, Y)$ | unwrap abstraction $\overline{X}$ and give $\overline{Y}$ to it |

$$\mathit{fct}[\![\lambda x.M]\!] =$$

**closure**(**abs**(

$\mathit{fct}[\![M]\!]$ ))

| **funcon** | **informal semantics** |
|---|---|
| **closure** (**abs** ($X$)) | yields **abs** (**close** (**scope** ($\Gamma, X$))) where $\Gamma$ is the current **environment** |
| **close** ($X$) | evaluate $X$ under the empty **environment** |

$$fct[\![\lambda x.M]\!] =$$
$$\mathbf{closure}(\mathbf{abs}(\mathbf{scope}(\mathbf{bind}(x, \mathbf{fst}(\mathbf{given}))$$
$$, \qquad fct[\![M]\!])\ ))$$

| funcon | informal semantics |
|---|---|
| **closure** (**abs** ($X$)) | yields **abs** (**close** (**scope** ($\Gamma, X$))) where $\Gamma$ is the current **environment** |
| **close** ($X$) | evaluate $X$ under the empty **environment** |

$$fct[\![\lambda x.M]\!] = \textbf{closure}(\textbf{abs}(\textbf{scope}(\textbf{bind}(x, \textbf{fst}(\textbf{given})) , \textbf{scope}(\textbf{bind}(\texttt{"this"}, \textbf{snd}(\textbf{given})), fct[\![M]\!]))))$$

| funcon | informal semantics |
| --- | --- |
| **closure** (**abs** ($X$)) | yields **abs** (**close** (**scope** ($\Gamma, X$))) where $\Gamma$ is the current **environment** |
| **close** ($X$) | evaluate $X$ under the empty **environment** |

$$\begin{aligned} \textit{fct}[\![\textbf{this}]\!] &= \textbf{bound}(\texttt{"this"}) \\ \textit{fct}[\![x]\!] &= \textbf{bound}(x) \quad \textbf{if } x \neq \textbf{this} \end{aligned}$$

| **funcon** | **informal semantics** |
|---|---|
| **bound** $(X)$ | yields $V$ if the current **environment** binds id $\overline{X}$ to $V$ |

Section 3

# Funcons for HGMP

# Research Questions

*Can we apply HGMPification to* Funcons*?*

a) HGMPification of Funcons
   i) Meta-reps of funcon terms (ASTs), with $\Downarrow_{dl}$ and $\Downarrow_{ul}$
   ii) Introduce a compilation phase for funcon terms
   iii) Compile-time HGMP: **meta-up**, **meta-down**, **meta-let**
   iv) Run-time HGMP: **meta-eval**

b) HGMPification of object language
   i) Translation for meta-programming constructs
   ii) Translation for meta-reps (ASTs)

# Meta-representations (ASTs)

Let **strings** represent funcon names

and $ty$ a function mapping a value $V$ to its type $\tau$ (**types**)

- New variadic funcon **ast** $(X_0, X_1, ..., X_k)$ with
  - $X_0$ (evaluates to) a funcon name or a type
  - $X_1, \ldots, X_k$ (evaluate to) the meta-reps of arguments
- New value constructor **astv** $(T, V_1, ..., V_k)$ with
  - If $T$ a type, then $k = 1$ and $V_1$ some value with $T = ty(V_1)$
  - If $T$ a funcon name, then $V_1, \ldots, V_k$ are **asts**

# Dynamic semantics of meta-representations

$$\frac{ty(V) = \tau}{\mathbf{ast}(\tau, V) \longrightarrow \mathbf{astv}(\tau, V)}$$

$$\frac{ty(T) = \mathbf{strings} \qquad ty(V_1) = \mathbf{asts} \ldots ty(V_n) = \mathbf{asts}}{\mathbf{ast}(T, V_1, \ldots, V_n) \longrightarrow \mathbf{astv}(T, V_1, \ldots, V_n)}$$

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

# Up meta-level

$$\frac{X_1 \Downarrow_{ul} X_1' \ldots X_n \Downarrow_{ul} X_n'}{\mathbf{funcon}_T(X_1, \ldots, X_n) \Downarrow_{ul} \mathbf{ast}(T, X_1', \ldots, X_n')}$$

$$\frac{}{V \Downarrow_{ul} \mathbf{astv}(ty(V), V)}$$

# Down meta-level

$$\frac{ty(T) = \textbf{strings} \qquad V_1 \Downarrow_{dl} X_1 \ldots V_k \Downarrow_{dl} X_k}{\textbf{astv}(T, V_1, \ldots, V_k) \Downarrow_{dl} \textbf{funcon}_T(X_1, \ldots, X_k)}$$

$$\frac{ty(\tau) = \textbf{types}}{\textbf{astv}(\tau, V) \Downarrow_{dl} V}$$

# Progress

a) HGMPification of FUNCONS
   i) Funcon term meta-reps (ASTs), with $\Downarrow_{dl}$ and $\Downarrow_{ul}$
   ii) Introduce a compilation phase for funcon terms
   iii) Compile-time HGMP: **meta-up**, **meta-down**, **meta-let**
   iv) Run-time HGMP: **meta-eval**

b) HGMPification of object language
   i) Translation for meta-programming constructs
   ii) Translation for meta-reps (ASTs)

# Progress

a) HGMPification of FUNCONS
   i) Funcon term meta-reps (ASTs), with $\Downarrow_{dl}$ and $\Downarrow_{ul}$
   ii) Introduce a compilation phase for funcon terms
      *How to combine with static semantics for* FUNCONS*?*
   iii) Compile-time HGMP: **meta-up**, **meta-down**, **meta-let**
   iv) Run-time HGMP: **meta-eval**

b) HGMPification of object language
   i) Translation for meta-programming constructs
   ii) Translation for meta-reps (ASTs)

# Progress

a) HGMPification of FUNCONS
   i) Funcon term meta-reps (ASTs), with $\Downarrow_{dl}$ and $\Downarrow_{ul}$
   ii) Introduce a compilation phase for funcon terms
      *How to combine with static semantics for* FUNCONS*?*
   iii) Compile-time HGMP: **meta-up**, **meta-down**, **meta-let**
      *Exactly as in Berger et al.*
   iv) Run-time HGMP: **meta-eval**

b) HGMPification of object language
   i) Translation for meta-programming constructs
   ii) Translation for meta-reps (ASTs)

# Progress

a) HGMPification of FUNCONS
   i) Funcon term meta-reps (ASTs), with $\Downarrow_{dl}$ and $\Downarrow_{ul}$
   ii) Introduce a compilation phase for funcon terms
      *How to combine with static semantics for* FUNCONS*?*
   iii) Compile-time HGMP: **meta-up**, **meta-down**, **meta-let**
      *Exactly as in Berger et al.*
   iv) Run-time HGMP: **meta-eval**
      *Exactly as in Berger et al.*

b) HGMPification of object language
   i) Translation for meta-programming constructs
   ii) Translation for meta-reps (ASTs)

# Translation of HGMP constructs

$$
\begin{aligned}
\mathit{fct}[\![\uparrow\{M\}]\!] &= \textbf{meta-up}(\mathit{fct}[\![M]\!]) \\
\mathit{fct}[\![\downarrow\{M\}]\!] &= \textbf{meta-down}(\mathit{fct}[\![M]\!]) \\
\mathit{fct}[\![\textbf{let}_{ct}\ x = M \ \textbf{in}\ N]\!] &= \textbf{meta-let}(x, \mathit{fct}[\![M]\!], \mathit{fct}[\![N]\!]) \\
\mathit{fct}[\![\mathit{eval}(M)]\!] &= \textbf{meta-eval}(\mathit{fct}[\![M]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fct}[\![\downarrow x]\!] &= \textbf{meta-down}(\mathit{fct}[\![M]\!]) \\
\mathit{fct}[\![\textbf{lift}\ M]\!] &= \textbf{give}(\mathit{fct}[\![M]\!], \textbf{ast}(\textbf{type-of}(\textbf{given}), \textbf{given}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fct}[\![\mathit{ast}_{app}(M, N)]\!] &= ??? \\
\ldots &= \ldots
\end{aligned}
$$

Recall translation of application:

$$\mathit{fct}[\![M\ N]\!] \;=\; \mathbf{give}(\mathit{fct}[\![M]\!], \mathbf{apply}(\mathbf{given}, (\mathit{fct}[\![N]\!], \mathbf{given})))$$

How do we translate the meta-rep of $\lambda$-application?

$$\mathit{fct}[\![\mathit{ast}_{\mathit{app}}(M, N)]\!] \;=\; \mathbf{ast}(\texttt{"give"}, \mathit{fct}[\![M]\!], \mathbf{ast}(\texttt{"apply"}, \ldots))$$

We have duplicated the translation of application...

## homomorphism property

*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 translations of application as follows

$$fct[\![M\ N]\!] \ = \ f_{app}(fct[\![M]\!], fct[\![N]\!])$$
$$\textbf{where } f_{app}(M, N) = \textbf{give}(M, \textbf{apply}(\textbf{given}, (N, \textbf{given})))$$

and the the translation of the meta-rep of application

$$fct[\![ast_{app}(M, N)]\!] \ = \textbf{meta-up}(f_{app}(\textbf{meta-down}(fct[\![M]\!]), \textbf{meta-down}(fct[\![N]\!])))$$

We introduce **ast-app** with the following dynamic semantics

$$\frac{V_1 \Downarrow_{dl} M \quad V_2 \Downarrow_{dl} N \qquad f_{app}(M, N) \Downarrow_{ul} F}{\textbf{ast-app}(V_1, V_2) \rightarrow F}$$

$$\frac{F_1 \rightarrow F_1'}{\textbf{ast-app}(F_1, F_2) \rightarrow \textbf{ast-app}(F_1', F_2)} \quad \frac{F_2 \rightarrow F_2'}{\textbf{ast-app}(F_1, F_2) \rightarrow \textbf{ast-app}(F_1, F_2')}$$

and translate the meta-rep of application directly into it

$$fct[\![ast_{app}(M, N)]\!] = \textbf{ast-app}(fct[\![M]\!], fct[\![N]\!])$$

- To complete the HGMPification of the $\lambda$-calculus:

$$\begin{aligned} \mathit{fct}[\![ \mathit{ast}_{\mathit{app}}(M, N) ]\!] &= \textbf{ast-app}(\mathit{fct}[\![ M ]\!], \mathit{fct}[\![ N ]\!]) \\ \mathit{fct}[\![ \mathit{ast}_{\mathit{lam}}(X, M) ]\!] &= \textbf{ast-lam}(\mathit{fct}[\![ X ]\!], \mathit{fct}[\![ M ]\!]) \\ \ldots &= \ldots \end{aligned}$$

- $\mathit{ast}_{\mathit{app}}(M, N)$ is concrete syntax determined by language design

$$\begin{aligned} \mathit{fct}[\![ \textbf{App}\ M\ N ]\!] &= \textbf{ast-app}(\mathit{fct}[\![ M ]\!], \mathit{fct}[\![ N ]\!]) \\ \mathit{fct}[\![ \textbf{Lambda}\ X\ M ]\!] &= \textbf{ast-lam}(\mathit{fct}[\![ X ]\!], \mathit{fct}[\![ M ]\!]) \\ \ldots &= \ldots \end{aligned}$$

# Conclusions

- Adding HGMP facilities to Funcons is relatively straightforward
- Adding object language ASTs risk duplication, but for homomorphic translations the process can be automated
- Potential benefits:
  - Languages with HGMP now in the scope of Funcons
  - Languages with effects in the scope of HGMP formalisation

# Future work

- More funcons: concurrency, unification (logic programming)
- Static semantics of funcons
- Further case studies, including languages with HGMP

*Does this work simplify giving a (component-based) semantics for languages with meta-programming facilities?*

# The Fundamental Constructs of Homogeneous Generative Meta-Programming

*or* Funcons for HGMP

L. Thomas van Binsbergen

Royal Holloway, University of London

17 January, 2018