Modelling homogeneous generative meta-programming

Martin Berger Laurence Tratt Christian Urban

"The formal model is quite similar to the interpreter proposed in McCarthy’s 1955 paper on LISP. I also didn’t find it particularly insightful" **ECOOP’17 review of this paper.**

"The formal model is quite similar to the interpreter proposed in McCarthy's 1955 paper on LISP. I also didn't find it particularly insightful" **ECOOP'17 review of this paper.**

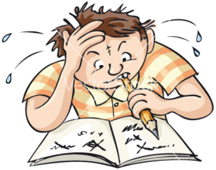
The reviewer is 95% right, but ...

Problem is a nutshell

List of industrial verification tools with first-class support for MP:

Problem is a nutshell

List of industrial verification tools with first-class support for MP:



What is MP?

Meta-programming = code as data.

What is MP?

Meta-programming = code as data.

Meta-programming: L-programs as data in L'.

What is MP?

Meta-programming = code as data.

Meta-programming: L-programs as data in L'.

Homogeneous meta-programming: MP where $L = L'$.

Why MP?

Lowering the “price of abstraction”, the hard trade-off between abstraction and performance, at the price of higher language complexity.

What is MP: example

```
printf( "System.out.println( \"Hello World!\" );" )
```

What is MP: example

```
printf( "System.out.println( \"Hello World!\" );" )
```

Meta-programming is simple if you don’t care about convenient, principled and safe handling of programs as data. Just use strings.

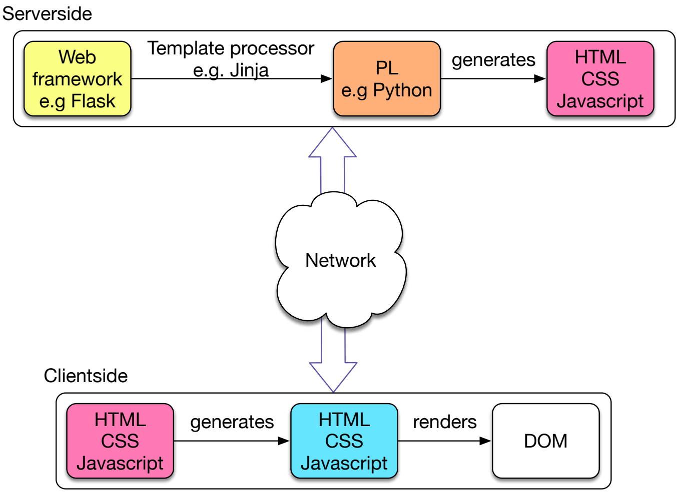
What is MP: example

```
printf( "System.out.println( \"Hello World!\" );" )
```

Meta-programming is simple if you don't care about convenient, principled and safe handling of programs as data. Just use strings.

Problem: strings contain 'junk'

MP is ubiquitous: example



Research hypothesis

What has been missing is a **simple** and **language independent** foundational approach towards MP that expresses the main dimensions of MP as first-class citizens on an equal basis, and shows how they interrelate.

Research hypothesis

$$\frac{\lambda\text{-calculus}}{\text{Functional programming}} = \frac{???}{\text{Meta-programming}}$$

Let’s simplify

We ignore:

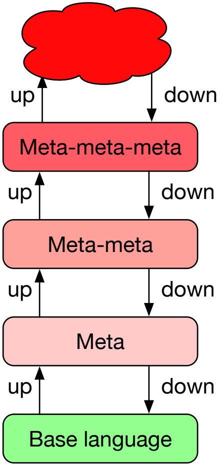
- Non-homogeneous meta-programming
- Hygiene
- Types
- Notions of equality
- Beauty of syntax
- Efficiency, performance
- Lexical rewriting (e.g. C preprocessor)
- ...

What we are looking for is a foundation that we use to study those later.

Research hypothesis

Essential features of MP:

- Language representation (code as data)
- Homogeneous meta-programming
- Language levels (base, meta, meta-meta ...)
- Navigation between language levels
- Computation is driven by the base-language



PL empiricism: the HGMP design space

PL empiricism: the HGMP design space

- What kind of MP?
- When is MP executed?
- How are programs represented as data?

HGMP design space: What kind of MP?

- **Homogeneous MP:** the object- and the meta-language are identical (examples: Racket, Template Haskell, MetaOcaml, Scala).
- In **heterogeneous MP** object- and the meta-language are different (example: C++ templates)

HGMP design space: What kind of MP?

- **Homogeneous MP:** the object- and the meta-language are identical (examples: Racket, Template Haskell, MetaOcaml, Scala).
- In **heterogeneous MP** object- and the meta-language are different (example: C++ templates)

We restrict our attention to homogeneous meta-programming.

HGMP design space: What kind of MP?

- **Generative MP:** where an program is generated (put together) by another program.
- **Intensional MP:** where an program is analysed (taken apart) by another program, e.g. reflection.

HGMP design space: What kind of MP?

- **Generative MP:** where an program is generated (put together) by another program.
- **Intensional MP:** where an program is analysed (taken apart) by another program, e.g. reflection.

Duality?

HGMP design space: What kind of MP?

- **Generative MP:** where an program is generated (put together) by another program.
- **Intensional MP:** where an program is analysed (taken apart) by another program, e.g. reflection.

Duality?

We restrict our attention to homogeneous generative meta-programming (HGMP).

HGMP design space: How are programs represented as data?

- Using strings.
- Abstract syntax trees (ASTs) typically using ADTs (algebraic data types).
- Quasi-quotes, where programs are represented by ’themselves’ (plus marker to distinguish code/data).

Reminder: quasi-quote

```
"I'm a quote"
```

Reminder: quasi-quote

`"I’m a quote"`

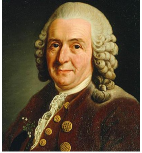
`"I’m a` $[\![(\lambda x.x)$ `"quasi"` $]\!]$ `quote"`

HGMP design space: How are programs represented as data?

Evaluation criteria:

- Syntactic overhead
- Support for generating only ’valid’ programs
- Expressivity

Taxonomy



Construct	Terse	only valid programs	expressive
Strings	●	○	●
ASTs	○	●	●
QQs	●	●	○

HGMP design space: How are programs represented as data?

Important goal: give both QQs and ASTs first class status, and show how they relate.

HGMP design space: When is MP executed?

- At **compile-time:** e.g. the Lisp family, Template Haskell, C++. We call this **CTMP**.
- At **run-time:** e.g. the MetaML family, JavaScript, printf-based MP. We call this **RTMP**.

HGMP design space: When is MP executed?

- At **compile-time:** e.g. the Lisp family, Template Haskell, C++. We call this **CTMP**.
- At **run-time:** e.g. the MetaML family, JavaScript, printf-based MP. We call this **RTMP**.

The difference is subtle. The result of CTMP is '**frozen**' (e.g. by saving the produced executable), multiple evaluations of a CTMP'ed program can be done with one compilation. RTMP'ed programs are **regenerated on every run**. Whether that leads to observable differences depends on the available language features.

HGMP design space: When is MP executed?

Important goal: give both CTMP and RTMP first class status, and show how they relate.

HGMP(λ) = λ-calculus with CTMP and RTMP

HGMP(λ) = λ-calculus with CTMP and RTMP

We start with the **untyped** λ-calculus, and CBV.

HGMP(λ) = λ-calculus with CTMP and RTMP

We start with the **untyped** λ-calculus, and CBV.

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

HGMP(λ) = λ-calculus with CTMP and RTMP

We start with the **untyped** λ-calculus, and CBV.

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

ASTs are the key representation of programs as data. So we add AST constructs for each element of the base language:

HGMP(λ) = λ-calculus with CTMP and RTMP

We start with the **untyped** λ-calculus, and CBV.

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

ASTs are the key representation of programs as data. So we add AST constructs for each element of the base language:

$$\begin{array}{rcl} M & ::= & ... \mid \mathrm{ast}_{\mathrm{t}}(\tilde{M}) \\ t & ::= & \mathrm{var} \mid \mathrm{app} \mid \mathrm{lam} \mid \mathrm{int} \mid \mathrm{string} \mid \mathrm{add} \mid ... \end{array}$$

Notice something?

Adding ASTs **mirrors** the syntax of the language. We make a 'copy' of the base language.

This is not λ-specific, we'd do the same for any other base.

HGMP(λ): adding CTMP

HGMP(λ): adding CTMP

We add **marker** to indicate compile-time HGMP should occur.

HGMP(λ): adding CTMP

We add **marker** to indicate compile-time HGMP should occur.

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

Meaning of ${\downarrow}\{M\}$ is

- M must be evaluated (= run) at compile-time
- CT-evaluation of M yields an AST
- AST gets 'spliced into' the rest of the AST the compiler is constructing
- Compilation proceeds

Operational semantics of the foundational calculus

We keep the usual $\Downarrow_\lambda$ from λ-calculus, but now add a second phase:

$$\underbrace{M \Downarrow_{ct}}_{\text{compile-time}} \; A \; \underbrace{\Downarrow_\lambda V}_{\text{run-time}}$$

$\Downarrow_{ct}$

Idea: $\Downarrow_{ct}$ scans for $\downarrow\{\cdot\}$ and eliminates them by evaluation and splicing.

$$\frac{}{x \Downarrow_{ct} x}\ \textsc{Var ct} \qquad \frac{M \Downarrow_{ct} A \quad N \Downarrow_{ct} B}{MN \Downarrow_{ct} AB}\ \textsc{App ct} \qquad \frac{M \Downarrow_{ct} N}{\lambda x.M \Downarrow_{ct} \lambda x.N}\ \textsc{Lam ct}$$

$$\frac{}{c \Downarrow_{ct} c}\ \textsc{Const ct} \qquad \frac{M \Downarrow_{ct} A \quad N \Downarrow_{ct} B}{M + N \Downarrow_{ct} A + B}\ \textsc{Add ct}$$

$$\frac{M_i \Downarrow_{ct} N_i}{\mathrm{ast}_{\mathrm{t}}(\tilde{M}) \Downarrow_{ct} \mathrm{ast}_{\mathrm{t}}(\tilde{N})}\ \textsc{Ast}_c\ \textsc{ct} \qquad \frac{M \Downarrow_{ct} A \quad A \Downarrow_{\lambda} B \quad B \Downarrow_{dl} C}{\downarrow\{M\} \Downarrow_{ct} C}\ \textsc{DownML ct}$$

$\Downarrow_{dl}$

Idea: $\Downarrow_{dl}$ removes one layer of ASTs, i.e. goes down a meta-level.

$$\frac{}{\mathsf{ast}_{\mathsf{var}}(''x'') \Downarrow_{dl} x}\ \textsc{Var dl} \qquad \frac{M \Downarrow_{dl} M' \quad N \Downarrow_{dl} N'}{\mathsf{ast}_{\mathsf{app}}(M, N) \Downarrow_{dl} M'N'}\ \textsc{App dl}$$

$$\frac{M \Downarrow_{dl} ''x'' \quad N \Downarrow_{dl} N'}{\mathsf{ast}_{\mathsf{lam}}(M, N) \Downarrow_{dl} \lambda x.N'}\ \textsc{Lam dl} \qquad \frac{}{\mathsf{ast}_{\mathsf{int}}(n) \Downarrow_{dl} n}\ \textsc{Int dl}$$

$$\frac{}{\mathsf{ast}_{\mathsf{string}}(''x'') \Downarrow_{dl} ''x''}\ \textsc{String dl} \qquad \frac{M \Downarrow_{dl} M' \quad N \Downarrow_{dl} N'}{\mathsf{ast}_{\mathsf{add}}(M, N) \Downarrow_{dl} M' + N'}\ \textsc{Add dl}$$

Note that non-ASTs have no $\Downarrow_{dl}$ rules, they are stuck.

Scoping

Our simple calculus intentionally allows variables to be captured dynamically, because **strings are not α-converted.**

Run-time HGMP

Run-time HGMP

It is now **easy** to add run-time HGMP:

Run-time HGMP

It is now **easy** to add run-time HGMP:

$$M ::= \ldots \mid \mathrm{eval}(M) \qquad\qquad t ::= \ldots \mid \mathrm{eval}$$

Run-time HGMP

It is now **easy** to add run-time HGMP:

$$M ::= \ldots \ \mid \ \mathrm{eval}(M) \qquad\qquad t ::= \ldots \ \mid \ \mathrm{eval}$$

We add the following rules to $\Downarrow_{ct}, \Downarrow_{\lambda}$ and $\Downarrow_{dl}$.

Run-time HGMP

It is now **easy** to add run-time HGMP:

$$M ::= \ldots \mid \mathrm{eval}(M) \qquad\qquad t ::= \ldots \mid \mathrm{eval}$$

We add the following rules to $\Downarrow_{ct}, \Downarrow_{\lambda}$ and $\Downarrow_{dl}$.

$$\frac{L \Downarrow_{\lambda} M \quad M \Downarrow_{dl} N \quad N \Downarrow_{\lambda} N'}{\mathrm{eval}(L) \Downarrow_{\lambda} N'}\ \textsc{Eval rt}$$

Enriching the calculus: higher-order ASTs

Enriching the calculus: higher-order ASTs

What about e.g. $\downarrow\{\downarrow\{M\}\}$, i.e. meta-meta-programming?

Calculus is extended with AST for ASTs, see paper for details.

Quasi-quotes

We have now finished, and obtained a λ-calculus with CTMP and RTMP.

But that calculus is lacks the convenience of quasi-quotes. Let's add them.

Quasi-quotes

ASTs are the cornerstone of our calculus.

For quasi-quotes, we extend the language:

$$M \quad ::= \quad ... \mid {\uparrow}\{M\}$$

Quasi-quotes

ASTs are the cornerstone of our calculus.

For quasi-quotes, we extend the language:

$$M \quad ::= \quad \dots \mid {\uparrow}\{M\}$$

We model QQs as “syntactic-sugar” to be removed at **compile-time** by conversion to ASTs, e.g.

$${\uparrow}\{2\} \quad \Downarrow_{ct} \quad \mathsf{ast}_{\mathsf{int}}(2)$$

Modelling the holes in QQs

Modelling the holes in QQs

Holes in quasi-quotes can run arbitrary computation. How to model that?

Modelling the holes in QQs

Holes in quasi-quotes can run arbitrary computation. How to model that?

Modelling the holes in QQs

Holes in quasi-quotes can run arbitrary computation. How to model that?

Let’s reuse $\downarrow\{\cdot\}$!

Modelling the holes in QQs

Holes in quasi-quotes can run arbitrary computation. How to model that?

Let's reuse $\downarrow\{\cdot\}$!

A downML $\downarrow\{\cdot\}$ inside $\uparrow\{\ldots \downarrow\{M\}\ldots\}$ is a 'hole' where arbitrary computation can be executed to produce an AST. This AST is then used as is. For example:

$$\uparrow\{2+7\} \quad \Downarrow_{ct} \quad \mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(7))$$

$$\uparrow\{2+\downarrow\{(\lambda x.x)\mathsf{ast}_{\mathsf{int}}(7)\}\} \quad \Downarrow_{ct} \quad \mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(7))$$

Operational semantics for $\uparrow\{M\}$

We introduce a new reduction relation $\Downarrow_{ul}$:

$$\frac{M \Downarrow_{ul} A}{\uparrow\{M\} \Downarrow_{ct} A}\ \textsc{UpML ct} \qquad \frac{M \Downarrow_{ct} A}{\downarrow\{M\} \Downarrow_{ul} A}\ \textsc{DownML ul}$$

$$\frac{}{''x'' \Downarrow_{ul} \mathrm{ast}_{\mathrm{string}}(''x'')}\ \textsc{String ul} \qquad \frac{M \Downarrow_{ul} A \quad N \Downarrow_{ul} B}{MN \Downarrow_{ul} \mathrm{ast}_{\mathrm{app}}(A, B)}\ \textsc{App ul}$$

$$\frac{M \Downarrow_{ul} A}{\lambda x.M \Downarrow_{ul} \mathrm{ast}_{\mathrm{lam}}(\mathrm{ast}_{\mathrm{string}}(''x''), A)}\ \textsc{Lam ul} \qquad \frac{}{\mathrm{tag}_{\mathrm{t}} \Downarrow_{ul} \mathrm{tag}_{\mathrm{t}}}\ \textsc{Tag ul}$$

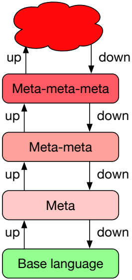
$$\frac{M \Downarrow_{ul} A}{\mathrm{eval}(M) \Downarrow_{ul} \mathrm{ast}_{\mathrm{eval}}(A)}\ \textsc{Eval ul} \qquad \frac{M \Downarrow_{ul} A \quad A \Downarrow_{ul} B}{\uparrow\{M\} \Downarrow_{ul} B}\ \textsc{UpML ul}$$

$$\frac{}{x \Downarrow_{ul} \mathrm{ast}_{\mathrm{var}}(''x'')}\ \textsc{Var ul} \qquad \frac{\ldots \quad M_i \Downarrow_{ul} A_i \quad \ldots}{\mathrm{ast}_{\mathrm{t}}(\tilde{M}) \Downarrow_{ul} \mathrm{ast}_{\mathrm{promote}}(\mathrm{tag}_{\mathrm{t}}, \tilde{A})}\ \textsc{Ast ul}$$

The rules capture our intuitions

The rules capture our intuitions

- $\uparrow\{\cdot\}$ goes up one meta-level (= adds a layer of ASTs).
- $\downarrow\{\cdot\}$ goes down one meta-level (= removes a layer of ASTs).



Thus RT-HGMP and CT-HGMP are connected as two facets of the same AST-coin.

$HGMP(\cdot)$

$HGMP(\cdot)$

Nothing in the HGMPification of λ-calculus depended on λ-calculus being the source language. The process was completely generic.

$HGMP(\cdot)$

$HGMP(\cdot)$

We seek to extend L with HGMP features to create L_{mp} . We can then create L_{mp} as follows:

HGMP$(\cdot)$

We seek to extend L with HGMP features to create L_{mp} . We can then create L_{mp} as follows:

- Mirror every syntactic element of L with an AST and a tag.

HGMP$(\cdot)$

We seek to extend L with HGMP features to create L_{mp} . We can then create L_{mp} as follows:

- Mirror every syntactic element of L with an AST and a tag.
- Add eval and tags eval and promote.

HGMP$(\cdot)$

We seek to extend *L* with HGMP features to create L_{mp} . We can then create L_{mp} as follows:

- Mirror every syntactic element of *L* with an AST and a tag.
- Add eval and tags eval and promote.
- Add $\uparrow\{\cdot\}$ and $\downarrow\{\cdot\}$.

HGMP$(\cdot)$

We seek to extend *L* with HGMP features to create L_{mp} . We can then create L_{mp} as follows:

- Mirror every syntactic element of *L* with an AST and a tag.
- Add eval and tags eval and promote.
- Add $\uparrow\{\cdot\}$ and $\downarrow\{\cdot\}$.

That gives us the syntax of L_{mp} . Operational semantics:

HGMP$(\cdot)$

We seek to extend L with HGMP features to create L_{mp} . We can then create L_{mp} as follows:

- Mirror every syntactic element of L with an AST and a tag.
- Add eval and tags eval and promote.
- Add $\uparrow\{\cdot\}$ and $\downarrow\{\cdot\}$.

That gives us the syntax of L_{mp} . Operational semantics:

Add reduction rules for ASTs, QQs and downMLs with computation driven by the base language. Note that *HGMP*(λ) does not change the reduction rules of λ-calculus itself. Note: only **adds** rules.

Thank you.