Foundations of meta-programming

Martin Berger

Cambridge, 9 August 2016

Based on joint work with Laurie Tratt and Christian Urban.

Research programme

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

Research programme

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

I want to convince you that the answer is **not** a calculus.

Research programme

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

I want to convince you that the answer is **not** a calculus.

To set the scene, let's remember why λ-calculus is good, and how meta-programming came about.

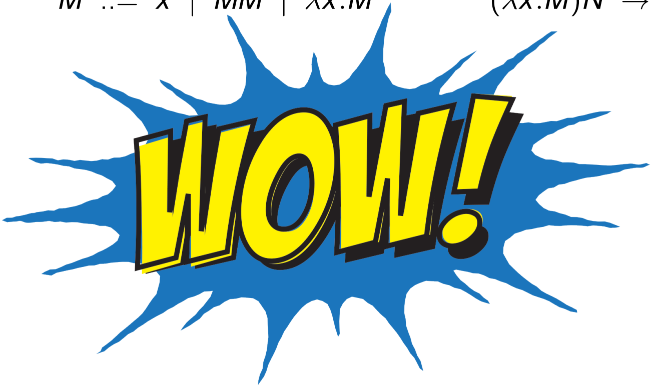
Why is λ-calculus so useful?

Why is λ-calculus so useful?

$$M ::= x \mid MM \mid \lambda x.M \qquad\qquad (\lambda x.M)N \to M[N/x]$$

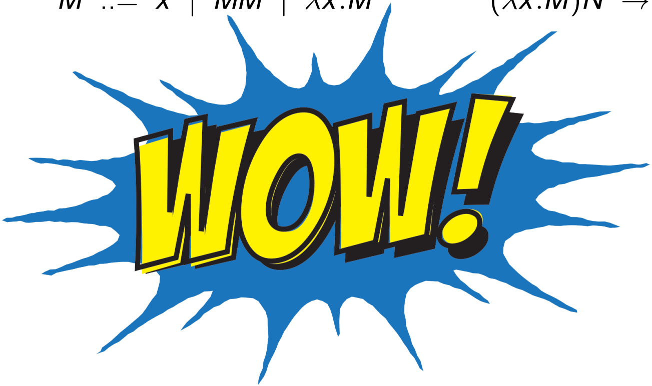
Why is λ-calculus so useful?

$$M ::= x \mid MM \mid \lambda x.M \qquad\qquad (\lambda x.M)N \to M[N/x]$$



Why is λ-calculus so useful?

$$M ::= x \mid MM \mid \lambda x.M \qquad\qquad (\lambda x.M)N \rightarrow M[N/x]$$



The point of theory is to **simplify**, to focus on essence.

Real-world programming languages:

Real-world programming languages:

- Strings
- Unicode
- FFI
- Backwards compatibility
- Modules
- Performance
- Ergonomics
- ...

In other words a real programming language is:

Let's do theory


META

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

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

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

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

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

Homogeneous generative meta-programming (HGMP) is the generation of programs by a program as the latter is being either compiled or executed.

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

Research on meta-programming is about **convenient**, **principled**, **general purpose** and **safe** handling of programs as data.

Research on meta-programming is about **convenient**, **principled**, **general purpose** and **safe** handling of programs as data.

But first ...

History

History

Quine invents quasi-quote and splicing (1940).

History

Quine invents quasi-quote and splicing (1940).

Lisp was the first language to support HGMP (1970s?).

History

Quine invents quasi-quote and splicing (1940).

Lisp was the first language to support HGMP (1970s?).

MetaML destroyed the persistent myth that HGMP works only on syntactically simple languages like Lisp (1990s).

History

Quine invents quasi-quote and splicing (1940).

Lisp was the first language to support HGMP (1970s?).

MetaML destroyed the persistent myth that HGMP works only on syntactically simple languages like Lisp (1990s).

Haskell might have been the first typed mainstream language with principled HGMP support (2002).

And then there is ...

And then there is ...

Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I[1]).

Von **Kurt Gödel** in Wien.

1.

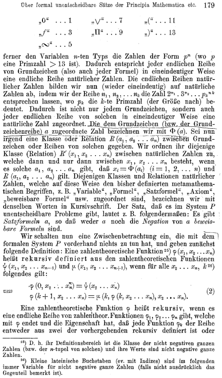
Die Entwicklung der Mathematik in der Richtung zu größerer Exaktheit hat bekanntlich dazu geführt, daß weite Gebiete von ihr formalisiert wurden, in der Art, daß das Beweisen nach einigen wenigen mechanischen Regeln vollzogen werden kann. Die umfassendsten derzeit aufgestellten formalen Systeme sind das System der Principia Mathematica (PM)[2]) einerseits, das Zermelo-Fraenkelsche (von J. v. Neumann weiter ausgebildete) Axiomensystem der Mengenlehre[3]) andererseits. Diese beiden Systeme sind so weit, daß alle heute in der Mathematik angewendeten Beweismethoden in ihnen formalisiert, d. h. auf einige wenige Axiome und Schlußregeln zurückgeführt sind. Es liegt daher die Vermutung nahe, daß diese Axiome und Schlußregeln dazu ausreichen, alle mathematischen Fragen, die sich in den betreffenden Systemen überhaupt formal ausdrücken

Arithmetisation of syntax

„0" . . . 1	„$\vee$" . . . 7	„(" . . . 11
„f" . . . 3	„Π" . . . 9	„)" . . . 13
„$\sim$" . . . 5		

ferner den Variablen n-ten Typs die Zahlen der Form p^n (wo p eine Primzahl > 13 ist). Dadurch entspricht jeder endlichen Reihe von Grundzeichen (also auch jeder Formel) in eineindeutiger Weise eine endliche Reihe natürlicher Zahlen. Die endlichen Reihen natürlicher Zahlen bilden wir nun (wieder eineindeutig) auf natürliche Zahlen ab, indem wir der Reihe $n_1, n_2, \ldots n_k$ die Zahl $2^{n_1} . 3^{n_2} \ldots p_k{}^{n_k}$ entsprechen lassen, wo p_k die k-te Primzahl (der Größe nach) bedeutet. Dadurch ist nicht nur jedem Grundzeichen, sondern auch jeder endlichen Reihe von solchen in eineindeutiger Weise eine natürliche Zahl zugeordnet. Die dem Grundzeichen (bzw. der Grundzeichenreihe) a zugeordnete Zahl bezeichnen wir mit $\Phi(a)$. Sei nun irgend eine Klasse oder Relation $R(a_1, a_2 \ldots a_n)$ zwischen Grundzeichen oder Reihen von solchen gegeben. Wir ordnen ihr diejenige Klasse (Relation) $R'(x_1, x_2 \ldots x_n)$ zwischen natürlichen Zahlen zu, welche dann und nur dann zwischen $x_1, x_2 \ldots x_n$ besteht, wenn es solche $a_1, a_2 \ldots a_n$ gibt, daß $x_i = \Phi(a_i)$ $(i = 1, 2, \ldots n)$ und $R(a_1, a_2 \ldots a_n)$ gilt. Diejenigen Klassen und Relationen natürlicher

Arithmetisation of syntax



Über formal unentscheidbare Sätze der Principia Mathematica etc. 179

„0" ... 1 „$\vee$" ... 7 „(" ... 11

„f" ... 3 „Π" ... 9 „)" ... 13

„$\sim$" ... 5

ferner den Variablen n-ten Typs die Zahlen der Form p^n (wo p eine Primzahl > 13 ist). Dadurch entspricht jeder endlichen Reihe von Grundzeichen (also auch jeder Formel) in eineindeutiger Weise eine endliche Reihe natürlicher Zahlen. Die endlichen Reihen natürlicher Zahlen bilden wir nun (wieder eineindeutig) auf natürliche Zahlen ab, indem wir der Reihe $n_1, n_2, \ldots n_k$ die Zahl $2^{n_1} . 3^{n_2} \ldots p_k^{n_k}$ entsprechen lassen, wo p_k die k-te Primzahl (der Größe nach) bedeutet. Dadurch ist nicht nur jedem Grundzeichen, sondern auch jeder endlichen Reihe von solchen in eineindeutiger Weise eine natürliche Zahl zugeordnet. Die dem Grundzeichen (bzw. der Grundzeichenreihe) a zugeordnete Zahl bezeichnen wir mit $\Phi(a)$. Sei nun irgend eine Klasse oder Relation $R(a_1, a_2 \ldots a_n)$ zwischen Grundzeichen oder Reihen von solchen gegeben. Wir ordnen ihr diejenige Klasse (Relation) $R'(x_1, x_2 \ldots x_n)$ zwischen natürlichen Zahlen zu, welche dann und nur dann zwischen $x_1, x_2 \ldots x_n$ besteht, wenn es solche $a_1, a_2 \ldots a_n$ gibt, daß $x_i = \Phi(a_i)$ $(i = 1, 2, \ldots n)$ und $R(a_1, a_2 \ldots a_n)$ gilt. Diejenigen Klassen und Relationen natürlicher Zahlen, welche auf diese Weise den bisher definierten metamathematischen Begriffen, z. B. „Variable", „Formel", „Satzformel", „Axiom", „beweisbare Formel" usw. zugeordnet sind, bezeichnen wir mit denselben Worten in Kursivschrift. Der Satz, daß es im System P unentscheidbare Probleme gibt, lautet z. B. folgendermaßen: Es gibt *Satzformeln* a, so daß weder a noch die *Negation* von a *beweisbare Formeln* sind.

Wir schalten nun eine Zwischenbetrachtung ein, die mit dem formalen System P vorderhand nichts zu tun hat, und geben zunächst folgende Definition: Eine zahlentheoretische Funktion[25] $\varphi(x_1, x_2 \ldots x_n)$ heißt **rekursiv definiert** aus den zahlentheoretischen Funktionen $\psi(x_1, x_2 \ldots x_{n-1})$ und $\mu(x_1, x_2 \ldots x_{n+1})$, wenn für alle $x_2 \ldots x_n$, k[26] folgendes gilt:

$$\begin{aligned} \varphi(0, x_2 \ldots x_n) &= \psi(x_2 \ldots x_n) \\ \varphi(k+1, x_2 \ldots x_n) &= \mu(k, \varphi(k, x_2 \ldots x_n), x_2 \ldots x_n). \end{aligned} \quad (2)$$

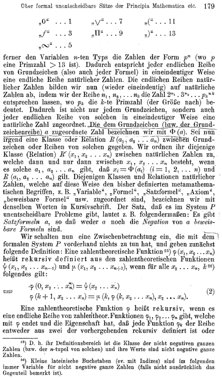
Eine zahlentheoretische Funktion φ heißt **rekursiv**, wenn es eine endliche Reihe von zahlentheor. Funktionen $\varphi_1, \varphi_2 \ldots \varphi_n$ gibt, welche mit φ endet und die Eigenschaft hat, daß jede Funktion φ_k der Reihe entweder aus zwei der vorhergehenden rekursiv definiert ist oder

[25]) D. h. ihr Definitionsbereich ist die Klasse der nicht negativen ganzen Zahlen (bzw. der n-tupel von solchen) und ihre Werte sind nicht negative ganze Zahlen.

[26]) Kleine lateinische Buchstaben (ev. mit Indizes) sind im folgenden immer Variable für nicht negative ganze Zahlen (falls nicht ausdrücklich das Gegenteil bemerkt ist).

Arithmetisation of syntax has been investigated in detail and led to powerful proof principles like logical reflection which strengthens the logic and computational reflection which makes proofs shorter.

Arithmetisation of syntax



Über formal unentscheidbare Sätze der Principia Mathematica etc. 179

„0" . . . 1	„$\vee$" . . . 7	„(" . . . 11
„f" . . . 3	„Π" . . . 9	„)" . . . 13
„$\sim$" . . . 5		

ferner den Variablen n-ten Typs die Zahlen der Form p^n (wo p eine Primzahl > 13 ist). Dadurch entspricht jeder endlichen Reihe von Grundzeichen (also auch jeder Formel) in eineindeutiger Weise eine endliche Reihe natürlicher Zahlen. Die endlichen Reihen natürlicher Zahlen bilden wir nun (wieder eineindeutig) auf natürliche Zahlen ab, indem wir der Reihe $n_1, n_2, \ldots n_k$ die Zahl $2^{n_1} \cdot 3^{n_2} \ldots p_k^{n_k}$ entsprechen lassen, wo p_k die k-te Primzahl (der Größe nach) bedeutet. Dadurch ist nicht nur jedem Grundzeichen, sondern auch jeder endlichen Reihe von solchen in eineindeutiger Weise eine natürliche Zahl zugeordnet. Die dem Grundzeichen (bzw. der Grundzeichenreihe) a zugeordnete Zahl bezeichnen wir mit $\Phi(a)$. Sei nun irgend eine Klasse oder Relation $R(a_1, a_2 \ldots a_n)$ zwischen Grundzeichen oder Reihen von solchen gegeben. Wir ordnen ihr diejenige Klasse (Relation) $R'(x_1, x_2 \ldots x_n)$ zwischen natürlichen Zahlen zu, welche dann und nur dann zwischen $x_1, x_2 \ldots x_n$ besteht, wenn es solche $a_1, a_2 \ldots a_n$ gibt, daß $x_i = \Phi(a_i)$ $(i = 1, 2, \ldots n)$ und $R(a_1, a_2 \ldots a_n)$ gilt. Diejenigen Klassen und Relationen natürlicher Zahlen, welche auf diese Weise den bisher definierten metamathematischen Begriffen, z. B. „Variable", „Formel", „Satzformel", „Axiom", „beweisbare Formel" usw. zugeordnet sind, bezeichnen wir mit denselben Worten in Kursivschrift. Der Satz, daß es im System P unentscheidbare Probleme gibt, lautet z. B. folgendermaßen: Es gibt *Satzformeln* a, so daß weder a noch die *Negation* von a *beweisbare Formeln* sind.

Wir schalten nun eine Zwischenbetrachtung ein, die mit dem formalen System P vorderhand nichts zu tun hat, und geben zunächst folgende Definition: Eine zahlentheoretische Funktion[25] $\varphi(x_1, x_2 \ldots x_n)$ heißt rekursiv definiert aus den zahlentheoretischen Funktionen $\psi(x_1, x_2 \ldots x_{n-1})$ und $\mu(x_1, x_2 \ldots x_{n+1})$, wenn für alle $x_2 \ldots x_n, k$[26] folgendes gilt:

$$\begin{aligned} \varphi(0, x_2 \ldots x_n) &= \psi(x_2 \ldots x_n) \\ \varphi(k+1, x_2 \ldots x_n) &= \mu(k, \varphi(k, x_2 \ldots x_n), x_2 \ldots x_n). \end{aligned} \qquad (2)$$

Eine zahlentheoretische Funktion φ heißt rekursiv, wenn es eine endliche Reihe von zahlentheor. Funktionen $\varphi_1, \varphi_2 \ldots \varphi_n$ gibt, welche mit φ endet und die Eigenschaft hat, daß jede Funktion φ_k der Reihe entweder aus zwei der vorhergehenden rekursiv definiert ist oder

[25]) D. h. ihr Definitionsbereich ist die Klasse der nicht negativen ganzen Zahlen (bzw. der n-tupel von solchen) und ihre Werte sind nicht negative ganze Zahlen.

[26]) Kleine lateinische Buchstaben (ev. mit Indizes) sind im folgenden immer Variable für nicht negative ganze Zahlen (falls nicht ausdrücklich das Gegenteil bemerkt ist).

Arithmetisation of syntax has been investigated in detail and led to powerful proof principles like logical reflection which strengthens the logic and computational reflection which makes proofs shorter.

Much work done by proof theorists and the dependent types community, connection with 'our' meta-programming not clear to me. See *J. Harrison*, *Metatheory and Reflection in Theorem Proving: A Survey and Critique* (1995) for more.

Meta-programming, circa August 2016

Meta-programming, circa August 2016

Most/all mainstream languages have some HGMP facilities, either as an upfront design decision (e.g. Scala, Rust, Javascript) or bolted on as the language evolved (e.g. C++).

Meta-programming, circa August 2016

Most/all mainstream languages have some HGMP facilities, either as an upfront design decision (e.g. Scala, Rust, Javascript) or bolted on as the language evolved (e.g. C++).

Working programmers heavily use HGMP, e.g. C++ generic programming, smart-pointers, DSL embedding. Syntax extension for increasing language expressivity, higher performance through compile- or run-time specialisation.

Meta-programming, circa August 2016

Most/all mainstream languages have some HGMP facilities, either as an upfront design decision (e.g. Scala, Rust, Javascript) or bolted on as the language evolved (e.g. C++).

Working programmers heavily use HGMP, e.g. C++ generic programming, smart-pointers, DSL embedding. Syntax extension for increasing language expressivity, higher performance through compile- or run-time specialisation.

In summary, meta-programming enables abstractions without run-time penalty. Thus MP resolves the tension between abstraction and performance, albeit **at the cost of increasing language complexity**.

Is it a solved problem in practise?

Is it a solved problem in practise?

Is it a solved problem in practise?

Scala just gutted it's original HGMP approach, to be replaced by `scala.meta`.

Is it a solved problem in practise?

Scala just gutted it’s original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

Is it a solved problem in practise?

Scala just gutted it's original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

MetaOcaml was found to be ... type-unsound.

Is it a solved problem in practise?

Scala just gutted it's original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

MetaOcaml was found to be ... type-unsound.

Template Haskell has meta-programming, but not meta-meta-programming etc.

Is it a solved problem in practise?

Scala just gutted it's original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

MetaOcaml was found to be ... type-unsound.

Template Haskell has meta-programming, but not meta-meta-programming etc.

Major terminological confusion, e.g. CTMP vs RTMP, macros vs CTMP.

Is it a solved problem in practise?

Scala just gutted it's original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

MetaOcaml was found to be ... type-unsound.

Template Haskell has meta-programming, but not meta-meta-programming etc.

Major terminological confusion, e.g. CTMP vs RTMP, macros vs CTMP.

Javascript's string-based MP is a security headache.

Is it a solved problem in practise?

Scala just gutted it's original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

MetaOcaml was found to be ... type-unsound.

Template Haskell has meta-programming, but not meta-meta-programming etc.

Major terminological confusion, e.g. CTMP vs RTMP, macros vs CTMP.

Javascript's string-based MP is a security headache.

Implementations of hygiene sometimes buggy or slow or unclear.

Is it a solved problem in practise?

Scala just gutted it’s original HGMP approach, to be replaced by `scala.meta`.

C++ template meta-programming is not pretty.

MetaOcaml was found to be ... type-unsound.

Template Haskell has meta-programming, but not meta-meta-programming etc.

Major terminological confusion, e.g. CTMP vs RTMP, macros vs CTMP.

Javascript’s string-based MP is a security headache.

Implementations of hygiene sometimes buggy or slow or unclear.

Tooling hard, e.g. debugging.

Is it a solved problem in theory?

Is it a solved problem in theory?

Is it a solved problem in theory?

No convincing theory of HGMP program equality.

Is it a solved problem in theory?

No convincing theory of HGMP program equality.

No convincing theory of HGMP types.

Is it a solved problem in theory?

No convincing theory of HGMP program equality.

No convincing theory of HGMP types.

No convincing way of **automatically** adding HGMP to a base language.

Is it a solved problem in theory?

No convincing theory of HGMP program equality.

No convincing theory of HGMP types.

No convincing way of **automatically** adding HGMP to a base language.

No convincing specification and reasoning (e.g. program logics) about HGMP.

Is it a solved problem in theory?

No convincing theory of HGMP program equality.

No convincing theory of HGMP types.

No convincing way of **automatically** adding HGMP to a base language.

No convincing specification and reasoning (e.g. program logics) about HGMP.

No convincing way of deriving semantics of embedded DSL from embedding

Not a solved problem, but ...

Not a solved problem, but ...

Good news 1: lot's of open problems.

Not a solved problem, but ...

Good news 1: lot's of open problems.

Good news 2: good solutions are immediately relevant for industry.

Not a solved problem, but ...

Good news 1: lot's of open problems.

Good news 2: good solutions are immediately relevant for industry.

Good news 3: problems look fairly tractable, no $P \stackrel{?}{=} NP$-like difficulties. Lot's of theory ready to go, e.g. nominal techniques, proof assistants.

Why are these problems still open?

Why are these problems still open?

META

Thinking about multiple levels of a language and their interactions is hard for humans.

Why are these problems still open?

Thinking about multiple levels of a language and their interactions is hard for humans.

`slash.dot.dot.at@at.dot.dotat.at`

Why are these problems still open?

Thinking about multiple levels of a language and their interactions is hard for humans.

```
slash.dot.dot.at@at.dot.dotat.at
```

Even humor can be based on this difficulty:

Why are these problems still open?

Thinking about multiple levels of a language and their interactions is hard for humans.

`slash.dot.dot.at@at.dot.dotat.at`

Even humor can be based on this difficulty:

All PL researchers are liars.

Why are these problems still open?

There is a huge need for MP since working programmers manipulate programs all the time, but ...

Why are these problems still open?

There is a huge need for MP since working programmers manipulate programs all the time, but ...

Why are these problems still open?

There is a huge need for MP since working programmers manipulate programs all the time, but ...

Adding HGMP is deceptively easy to the untrained eye ...

Why are these problems still open?

There is a huge need for MP since working programmers manipulate programs all the time, but ...

Adding HGMP is deceptively easy to the untrained eye ... just add a data type representing programs ...

How hard can it be?

How hard can it be?

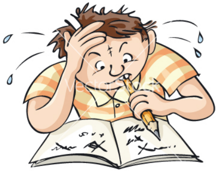
How hard can it be?

Remember: real programming languages are a mess

How hard can it be?

Remember: real programming languages are a mess

Adding HGMP to mess, means we need to create meta-mess, meta-meta-mess ... and think about how mess, meta-mess, meta-meta-mess ... relate.



Time for theory

Let’s simplify

Let's simplify

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

Let's simplify

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

... that means we focus on the **essential features** of HGMP, and **nothing else.**

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

Let's simplify

We ignore:

- Hygiene
- Types
- Notions of equality
- Beauty of syntax
- Efficiency, performance
- ...

Let's simplify

We ignore:

- Hygiene
- Types
- Notions of equality
- Beauty of syntax
- Efficiency, performance
- ...

But:

Let's simplify

We ignore:

- Hygiene
- Types
- Notions of equality
- Beauty of syntax
- Efficiency, performance
- ...

But: **What base language?**

What base language?

The PL research community will likely say λ-calculus.

What base language?

The PL research community will likely say λ-calculus.

This is not a bad choice.

What base language?

The PL research community will likely say λ-calculus.

This is not a bad choice.

What we really want is to add HGMP to **arbitrary** base languages.

What base language?

The PL research community will likely say λ-calculus.

This is not a bad choice.

What we really want is to add HGMP to **arbitrary** base languages. I.e. an explicit **function** $HGMP(\cdot)$:

$$L \mapsto L'$$

taking a programming language L as input and returning as output a language L' that is the HGMPified version of L.

Why arbitrary base language?

Why arbitrary base language?

Remember real-world programming languages:

Why arbitrary base language?

Remember real-world programming languages:

We want the transition

$$\text{mess} \rightarrow \text{meta-mess}$$

automatised ...

Why arbitrary base language?

Remember real-world programming languages:

We want the transition

$$\text{mess} \to \text{meta-mess}$$

automatised ...

... so we can experiment with languages without being drowned in uninteresting minutiae.

Research hypothesis

The foundation of meta-programming is the **function** *HGMP*$(\cdot)$:

$$\frac{\lambda\text{-calculus}}{\text{Functional programming}} = \frac{\color{red}{\mathit{HGMP}(\cdot)}}{\text{Meta-programming}}$$

Note that $HGMP(\cdot)$ is a theoretical tool, it's not intended to give nice result, e.g. good looking syntax.

But ...

Does *HGMP*$(\cdot)$ exist at all (as a computable algorithm)?

But ...

Does $HGMP(\cdot)$ exist at all (as a computable algorithm)?

If $HGMP(\cdot)$ exists, is it generic in the choice of base language, or highly dependent on the details of base language?

But ...

Does *HGMP*$(\cdot)$ exist at all (as a computable algorithm)?

If *HGMP*$(\cdot)$ exists, is it generic in the choice of base language, or highly dependent on the details of base language?

Precise answers to these questions are not yet known, but I want to sketch why I think *HGMP*$(\cdot)$ is essentially generic in the choice of base.

But ...

Does $\mathit{HGMP}(\cdot)$ exist at all (as a computable algorithm)?

If $\mathit{HGMP}(\cdot)$ exists, is it generic in the choice of base language, or highly dependent on the details of base language?

Precise answers to these questions are not yet known, but I want to sketch why I think $\mathit{HGMP}(\cdot)$ is essentially generic in the choice of base.

I will look at the special case of $\mathit{HGMP}(\lambda)$, and then generalise.

But ...

Does $\mathit{HGMP}(\cdot)$ exist at all (as a computable algorithm)?

If $\mathit{HGMP}(\cdot)$ exists, is it generic in the choice of base language, or highly dependent on the details of base language?

Precise answers to these questions are not yet known, but I want to sketch why I think $\mathit{HGMP}(\cdot)$ is essentially generic in the choice of base.

I will look at the special case of $\mathit{HGMP}(\lambda)$, and then generalise.

But first: PL empiricism: the HGMP design space

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, Converge).
- In **heterogeneous MP** object- and the meta-language are different (example: compiler written in C from Java to x86).

HGMP design space: What kind of MP?

- **Homogeneous MP:** the object- and the meta-language are identical (examples: Racket, Template Haskell, MetaOcaml, Converge).
- In **heterogeneous MP** object- and the meta-language are different (example: compiler written in C from Java to x86).

We restrict our attention to homogeneous meta-programming.

HGMP design space: What kind of MP?

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

HGMP design space: What kind of MP?

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

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

HGMP design space: How are programs represented as data?

- Using strings.
- ADTs (algebraic data types).
- Higher-level language support, e.g. upMLs, downMLs, quasi-quotes, inserts and splices.

HGMP design space: How are programs represented as data?

- Using strings.
- ADTs (algebraic data types).
- Higher-level language support, e.g. upMLs, downMLs, quasi-quotes, inserts and splices.

Let's look at them briefly. Consider the following criteria:

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

Strings

Strings

Example, selector function that chooses the *i*-the component from an *n*-tuple.

```
let pi_1_0 =  fun ( x ) -> x;;
let pi_2_0 =  fun ( x, _  ) -> x;;
let pi_2_1 =  fun ( _, x ) -> x;;
let pi_3_0 =  fun ( x, _, _  ) -> x;;
let pi_3_1 =  fun ( _, x, _  ) -> x;;
let pi_3_2 =  fun ( _, _, x ) -> x;;
let pi_4_0 =  fun ( x, _, _, _  ) -> x;;
let pi_4_1 =  fun ( _, x, _, _  ) -> x;;
let pi_4_2 =  fun ( _, _, x, _  ) -> x;;
let pi_4_3 =  fun ( _, _, _, x ) -> x;;
let pi_5_0 =  fun ( x, _, _, _, _  ) -> x;;
let pi_5_1 =  fun ( _, x, _, _, _  ) -> x;;
let pi_5_2 =  fun ( _, _, x, _, _  ) -> x;;
let pi_5_3 =  fun ( _, _, _, x, _  ) -> x;;
let pi_5_4 =  fun ( _, _, _, _, x ) -> x;;
```

selector.ml

Advantages of string-based MP

- Flexible, expressive.
- Easy to do for basic MP.
- Ubiquitous support.
- Not restricted to a single target language.

Disadvantages of string-based MP

- No language support for constructing syntactically correct programs.
- No support for “hygiene”, i.e. sane management of free and bound variables. Hygiene is hard to add for strings.

Disadvantages of string-based MP

- No language support for constructing syntactically correct programs.
- No support for “hygiene”, i.e. sane management of free and bound variables. Hygiene is hard to add for strings.

NB Lack of hygiene is sometimes useful, especially in large-scale MP.

Disadvantages of string-based MP

- No language support for constructing syntactically correct programs.
- No support for “hygiene”, i.e. sane management of free and bound variables. Hygiene is hard to add for strings.

NB Lack of hygiene is sometimes useful, especially in large-scale MP.

We reject strings in our foundational approach.

ADTs

ADTs

```
sealed abstract class Binop
   case class Add () extends Binop
   case class Sub () extends Binop
   case class Mul () extends Binop
   case class Div () extends Binop
   case class Eq () extends Binop

sealed abstract class Term
   case class CInt ( n : Int ) extends Term
   case class Op2 ( m : Term, op : Binop, n : Term ) ext
   case class Var ( x : Int ) extends Term
   case class App ( m : Term, n : Term ) extends Term
   case class Lam ( x : Int, m : Term ) extends Term
   case class Rec ( f : Int, x : Int, m : Term ) extends
   case class If ( c : Term, m : Term, n : Term ) extend
```

We call this representation AST (abstract syntax tree), the workhorse of HGMP.

Advantages and disadvantages of ADTs

Advantages and disadvantages of ADTs

Advantage:

Advantages and disadvantages of ADTs

Advantage:

- ADTs only construction of syntactically valid programs (may fail to type-check).

Disadvantages:

Advantages and disadvantages of ADTs

Advantage:

- ADTs only construction of syntactically valid programs (may fail to type-check).

Disadvantages:

- Verbose.

Advantages and disadvantages of ADTs

Advantage:

- ADTs only construction of syntactically valid programs (may fail to type-check).

Disadvantages:

- Verbose.
- No support for “hygiene”, i.e. sane management of free and bound variables. But hygiene is easy to add.

What we really want is ...

What we really want is ...

... to combine the terseness of strings with the guarantees of syntactic correctness that ASTs offer.


META

UpMLs and downML

UpMLs and downML

Enter **upMLs** and **downMLs**, another good idea from logic, first introduced in Lisp.

UpMLs (AKA quasi-quotes, backquotes) are quotes with holes. In the holes we can execute arbitrary programs that produces code.

DownMLs (AKA splices or inserts) are the corresponding un-quotation mechanism.

UpMLs and downML

UpMLs and downML

UpML (Up MetaLevel) represent AST (or AST-like) structures by quoted chunks of normal program syntax. We have chosen the term 'upMLs' to highlight an important relationship with downMLs.

UpMLs and downML

UpML (Up MetaLevel) represent AST (or AST-like) structures by quoted chunks of normal program syntax. We have chosen the term 'upMLs' to highlight an important relationship with downMLs.

$$\uparrow\{2+3\}$$

UpMLs and downML

UpML (Up MetaLevel) represent AST (or AST-like) structures by quoted chunks of normal program syntax. We have chosen the term ’upMLs’ to highlight an important relationship with downMLs.

$$\uparrow\{2 + 3\}$$

is short for

UpMLs and downML

UpML (Up MetaLevel) represent AST (or AST-like) structures by quoted chunks of normal program syntax. We have chosen the term 'upMLs' to highlight an important relationship with downMLs.

$$\uparrow\{2 + 3\}$$

is short for

$$\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$$

UpMLs and downML

UpML (Up MetaLevel) represent AST (or AST-like) structures by quoted chunks of normal program syntax. We have chosen the term 'upMLs' to highlight an important relationship with downMLs.

$$\uparrow\{2 + 3\}$$

is short for

$$\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$$

UpMLs are “syntactic sugar” for ASTs.

UpMLs and downML

UpMLs and downML

DownML (Down MetaLevel) indicate “holes” in upMLs. Inside the holes, arbitrary computation takes place during the evaluation of an upML. That computation **must** yield an AST.

UpMLs and downML

DownML (Down MetaLevel) indicate “holes” in upMLs. Inside the holes, arbitrary computation takes place during the <u>evaluation</u> of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathrm{ast}_{\mathrm{int}}(x - 1)$:

UpMLs and downML

DownML (Down MetaLevel) indicate "holes" in upMLs. Inside the holes, arbitrary computation takes place during the evaluation of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathrm{ast}_{\mathrm{int}}(x - 1)$:

$$\uparrow\{2+ \downarrow\{M\ 4\}\}$$

UpMLs and downML

DownML (Down MetaLevel) indicate "holes" in upMLs. Inside the holes, arbitrary computation takes place during the evaluation of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathsf{ast}_{\mathsf{int}}(x - 1)$:

$$\uparrow\{2 + \downarrow\{M\ 4\}\}$$

$$\uparrow\{2 + \downarrow\{\mathsf{ast}_{\mathsf{int}}(4 - 1)\}\}$$

UpMLs and downML

DownML (Down MetaLevel) indicate "holes" in upMLs. Inside the holes, arbitrary computation takes place during the <u>evaluation</u> of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathsf{ast}_{\mathsf{int}}(x-1)$:

$$\uparrow\{2 + \downarrow\{M\ 4\}\}$$

$$\uparrow\{2 + \downarrow\{\mathsf{ast}_{\mathsf{int}}(4-1)\}\}$$

$$\uparrow\{2 + \downarrow\{\mathsf{ast}_{\mathsf{int}}(3)\}\}$$

UpMLs and downML

DownML (Down MetaLevel) indicate “holes” in upMLs. Inside the holes, arbitrary computation takes place during the evaluation of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathsf{ast}_{\mathsf{int}}(x - 1)$:

$$\uparrow\{2+\downarrow\{M\ 4\}\}$$

$$\uparrow\{2+\downarrow\{\mathsf{ast}_{\mathsf{int}}(4-1)\}\}$$

$$\uparrow\{2+\downarrow\{\mathsf{ast}_{\mathsf{int}}(3)\}\}$$

$$\uparrow\{2+3\}$$

UpMLs and downML

DownML (Down MetaLevel) indicate “holes” in upMLs. Inside the holes, arbitrary computation takes place during the evaluation of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathrm{ast}_{\mathrm{int}}(x-1)$:

$$\uparrow\{2+\downarrow\{M\ 4\}\}$$

$$\uparrow\{2+\downarrow\{\mathrm{ast}_{\mathrm{int}}(4-1)\}\}$$

$$\uparrow\{2+\downarrow\{\mathrm{ast}_{\mathrm{int}}(3)\}\}$$

$$\uparrow\{2+3\}$$

$$\mathrm{ast}_{\mathrm{add}}(\mathrm{ast}_{\mathrm{int}}(2), \mathrm{ast}_{\mathrm{int}}(3))$$

UpMLs and downML

DownML (Down MetaLevel) indicate "holes" in upMLs. Inside the holes, arbitrary computation takes place during the <u>evaluation</u> of an upML. That computation **must** yield an AST. Exampe: with $M = \lambda x.\mathsf{ast}_{\mathsf{int}}(x-1)$:

$$\uparrow\{2+\downarrow\{M\ 4\}\}$$

$$\uparrow\{2+\downarrow\{\mathsf{ast}_{\mathsf{int}}(4-1)\}\}$$

$$\uparrow\{2+\downarrow\{\mathsf{ast}_{\mathsf{int}}(3)\}\}$$

$$\uparrow\{2+3\}$$

$$\mathsf{ast}_{\mathsf{add}}(\mathsf{ast}_{\mathsf{int}}(2), \mathsf{ast}_{\mathsf{int}}(3))$$

NB: DownMLs make sense only within the process of AST generation.

UpML and DownML in Racket

UpML and DownML in Racket

```
Welcome to Racket v6.5.
> (quasiquote (+ 2 3))
'(+ 2 3)

> (eval (quasiquote (+ 2 3)))
5

> (define f  (lambda (x) (quasiquote (* (unquote x) (unquote x)))))

> (f 5)
'(* 5 5)

> (eval (f 5))
25

>  (define g (lambda (x) (quasiquote (* 8 (unquote (f x))))))

> (g 3)
'(* 8 (* 3 3))

> (eval (g 3))
72
```

racket.r

UpMLs and hygiene

UpMLs and hygiene

By default upMLs in Racket, MetaOCaml, Converge and other languages are hygienic, i.e. prevent capture of free variables. This is an implementation choice, but not necessary. Advanced MP languages like Racket or Converge offer both, capturing and non-capturing behaviour.

UpMLs and hygiene

By default upMLs in Racket, MetaOCaml, Converge and other languages are hygienic, i.e. prevent capture of free variables. This is an implementation choice, but not necessary. Advanced MP languages like Racket or Converge offer both, capturing and non-capturing behaviour.

We will strictly separate both concepts, i.e. our upMLs are not hygienic.

HGMP design space: When is MP executed?

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

Some languages support both (e.g. Converge, `scala.meta`).

HGMP design space: When is MP executed?

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

Some languages support both (e.g. Converge, `scala.meta`).

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?

Example.

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

HGMP design space: When is MP executed?

Example.

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

Evaluates code at **run-time** using eval and prints ...

HGMP design space: When is MP executed?

Example.

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

Evaluates code at **run-time** using eval and prints ... `1 3 6`

HGMP design space: When is MP executed?

Example.

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

Evaluates code at **run-time** using eval and prints ... `1 3 6`

Replacing the eval with a downML to get **compile-time** evaluation:

```
print(1);
print(2 + ↓{print(3); ASTInt(4)})
```

yields ...

HGMP design space: When is MP executed?

Example.

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

Evaluates code at **run-time** using eval and prints ... `1 3 6`

Replacing the eval with a downML to get **compile-time** evaluation:

```
print(1);
print(2 + ↓{print(3); ASTInt(4)})
```

yields ... `3 1 6`

The `3` is printed during compilation, the `1 6` is printed every time the compiled code is run.

Modern languages and HGMP

Modern languages and HGMP

Language	Strings	ASTs	UpMLs	CT-HGMP	RT-HGMP
Converge	●	●	●	●	●
JavaScript	●	○	○	○	●
Lisp	●	●	●	●	●
MetaML	○	○	●	○	●
Haskell	○	●	●	●	○
Scala	○	●	●	●	●

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.

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

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.

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

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

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

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

META

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:

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

An AST constructor $\mathsf{ast}_{\mathsf{t}}(\tilde{M})$ takes $|M|+1$ arguments. **Tag** t specifies the specific AST datatype. The rest is relative to that datatype.

HGMP(λ): examples

HGMP(λ): examples

$\text{ast}_{\text{var}}(''x'')$ is the AST representation of the variable x

HGMP(λ): examples

$\text{ast}_{\text{var}}(''x'')$ is the AST representation of the variable x

$\text{ast}_{\text{int}}(3)$ is the AST representation of the constant 3

HGMP(λ): examples

$\text{ast}_{\text{var}}(''x'')$ is the AST representation of the variable x

$\text{ast}_{\text{int}}(3)$ is the AST representation of the constant 3

$\text{ast}_{\text{lam}}(\text{ast}_{\text{string}}(''x''), \text{ast}_{\text{var}}(''x''))$ is the AST of $\lambda x.x$

Notice something?

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 downMLs, to indicate compile-time HGMP should occur.

HGMP(λ): adding CTMP

We add downMLs, to indicate compile-time HGMP should occur.

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

HGMP(λ): adding CTMP

We add downMLs, to indicate compile-time HGMP should occur.

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

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

HGMP(λ): adding CTMP

We add downMLs, to indicate compile-time HGMP should occur.

$$M \quad ::= \quad ... \mid \downarrow\{M\} \qquad\qquad t \quad ::= \quad ...$$

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

- M must be evaluated (= run) at compile-time

HGMP(λ): adding CTMP

We add downMLs, to indicate compile-time HGMP should occur.

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

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

- M must be evaluated (= run) at compile-time
- CT-evaluation of M yields an AST

HGMP(λ): adding CTMP

We add downMLs, to indicate compile-time HGMP should occur.

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

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

HGMP(λ): adding CTMP

We add downMLs, to indicate compile-time HGMP should occur.

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

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:

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

Operational semantics of the foundational calculus

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

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

How does $\Downarrow_{ct}$ work?

Operational semantics of the foundational calculus

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

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

How does $\Downarrow_{ct}$ work?

- $\Downarrow_{ct}$ 'searches' through code for $\downarrow\{M\}$ to eliminate them.

Operational semantics of the foundational calculus

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

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

How does $\Downarrow_{ct}$ work?

- $\Downarrow_{ct}$ 'searches' through code for $\downarrow\{M\}$ to eliminate them.
- For every $\downarrow\{M\}$ is found:

Operational semantics of the foundational calculus

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

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

How does $\Downarrow_{ct}$ work?

- $\Downarrow_{ct}$ 'searches' through code for $\downarrow\{M\}$ to eliminate them.
- For every $\downarrow\{M\}$ is found:
  - M is recursively scanned for downMLs, yielding M'

Operational semantics of the foundational calculus

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

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

How does $\Downarrow_{ct}$ work?

- $\Downarrow_{ct}$ 'searches' through code for $\downarrow\{M\}$ to eliminate them.
- For every $\downarrow\{M\}$ is found:
  - M is recursively scanned for downMLs, yielding M'
  - then M' is evaluated using $\Downarrow_\lambda$, the usual CBV evaluation of λ-calculus, giving an AST A

Operational semantics of the foundational calculus

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

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

How does $\Downarrow_{ct}$ work?

- $\Downarrow_{ct}$ 'searches' through code for $\downarrow\{M\}$ to eliminate them.
- For every $\downarrow\{M\}$ is found:
  - M is recursively scanned for downMLs, yielding M'
  - then M' is evaluated using $\Downarrow_\lambda$, the usual CBV evaluation of λ-calculus, giving an AST A
  - Then $\Downarrow_{dl}$ de-ASTifies A, and splice into rest of program

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

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

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

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

$\Downarrow_{dl}$

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

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

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

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

$\Downarrow_{dl}$

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

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

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

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

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

Scoping

Scoping

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

Scoping

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

- $\lambda x. \downarrow\{\mathrm{ast}_{\mathrm{var}}(''x'')\} \Downarrow_{ct} \lambda x.x$.

Scoping

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

- $\lambda x.\ \downarrow\{\text{ast}_{\text{var}}(\text{"}x\text{"})\} \Downarrow_{ct} \lambda x.x$.
- $\lambda y.\ \downarrow\{\text{ast}_{\text{var}}(\text{"}x\text{"})\} \Downarrow_{ct} \lambda y.x$.

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 $\Downarrow_{dl}$.

$$\frac{M \Downarrow_{ct} N}{\mathrm{eval}(M) \Downarrow_{ct} \mathrm{eval}(N)}\ \textsc{Eval ct} \qquad \frac{M \Downarrow_{dl} N}{\mathrm{ast}_{\mathrm{eval}}(M) \Downarrow_{dl} \mathrm{eval}(N)}\ \textsc{Eval dl}$$

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

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 $\Downarrow_{dl}$.

$$\frac{M \Downarrow_{ct} N}{\mathrm{eval}(M) \Downarrow_{ct} \mathrm{eval}(N)}\ \textsc{Eval ct} \qquad \frac{M \Downarrow_{dl} N}{\mathrm{ast}_{\mathrm{eval}}(M) \Downarrow_{dl} \mathrm{eval}(N)}\ \textsc{Eval dl}$$

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

Note that eval is **not** 'disappeared' at compile-time.

Enriching the calculus: higher-order ASTs

Enriching the calculus: higher-order ASTs

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

Enriching the calculus: higher-order ASTs

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

Calculus so far doesn't allow the representation of ASTs as ASTs.

Enriching the calculus: higher-order ASTs

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

Calculus so far doesn't allow the representation of ASTs as ASTs.

This can be handled in several ways, we 'internalise' tags:

Enriching the calculus: higher-order ASTs

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

Calculus so far doesn't allow the representation of ASTs as ASTs.

This can be handled in several ways, we 'internalise' tags:

$$M ::= \ldots \mid \mathsf{tag}_t \qquad\qquad t ::= \ldots \mid \mathsf{promote}$$

Hence AST datatype $\mathsf{ast}_{\mathsf{promote}}(M, \tilde{N})$ which allows an arbitrary AST with a tag M and parameters $\tilde{N}$ to be promoted up a meta-level. Promoted ASTs can then be reduced one meta-level with the existing $\Downarrow_{dl}$ relation. E.g.:

Enriching the calculus: higher-order ASTs

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

Calculus so far doesn't allow the representation of ASTs as ASTs.

This can be handled in several ways, we 'internalise' tags:

$$M ::= \ldots \mid \mathsf{tag}_t \qquad\qquad t ::= \ldots \mid \mathsf{promote}$$

Hence AST datatype $\mathsf{ast}_{\mathsf{promote}}(M, \tilde{N})$ which allows an arbitrary AST with a tag M and parameters $\tilde{N}$ to be promoted up a meta-level. Promoted ASTs can then be reduced one meta-level with the existing $\Downarrow_{dl}$ relation. E.g.:

$$\mathsf{ast}_{\mathsf{promote}}(\mathsf{string}, \mathsf{ast}_{\mathsf{string}}("x")) \Downarrow_{dl} \mathsf{ast}_{\mathsf{string}}("x")$$

Operational semantics of higher-order ASTs

Operational semantics of higher-order ASTs

$$\frac{}{\mathsf{tag}_{\mathsf{t}} \Downarrow_{dl} \mathsf{tag}_{\mathsf{t}}}\ \textsc{Promote Tag}$$

$$\frac{L \Downarrow_{dl} \mathsf{tag}_{\mathsf{t}} \quad t \neq \mathsf{promote} \quad \ldots \quad M_i \Downarrow_{dl} N_i \quad \ldots}{\mathsf{ast}_{\mathsf{promote}}(L, \tilde{M}) \Downarrow_{dl} \mathsf{ast}_{\mathsf{t}}(\tilde{N})}\ \textsc{Promote dl 1}$$

$$\frac{L \Downarrow_{dl} \mathsf{tag}_{\mathsf{promote}} \quad M \Downarrow_{dl} \mathsf{tag}_{\mathsf{t}} \quad \ldots \quad N_i \Downarrow_{dl} N_i' \quad \ldots}{\mathsf{ast}_{\mathsf{promote}}(L, M, \tilde{N}) \Downarrow_{dl} \mathsf{ast}_{\mathsf{promote}}(\mathsf{tag}_{\mathsf{t}}, \tilde{N}')}\ \textsc{Promote dl 2}$$

$\Downarrow_{\lambda}$ and $\Downarrow_{ct}$ are unchanged as rules, but work on larger set of programs.

UpMLs (aka quasi-quotes)

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

UpMLs (aka quasi-quotes)

UpMLs (aka quasi-quotes)

ASTs are the cornerstone of our calculus. But ASTs are verbose. UpMLs (aka quasi-quotes or back-quotes) ameliorate this problem by allowing concrete syntax to be used to represent ASTs.

UpMLs (aka quasi-quotes)

ASTs are the cornerstone of our calculus. But ASTs are verbose. UpMLs (aka quasi-quotes or back-quotes) ameliorate this problem by allowing concrete syntax to be used to represent ASTs.

To add UpMLs to our language, we first extend the grammar as follows:

UpMLs (aka quasi-quotes)

ASTs are the cornerstone of our calculus. But ASTs are verbose. UpMLs (aka quasi-quotes or back-quotes) ameliorate this problem by allowing concrete syntax to be used to represent ASTs.

To add UpMLs to our language, we first extend the grammar as follows:

$$M \quad ::= \quad \ldots \mid \uparrow\{M\} \qquad\qquad t \quad ::= \quad \ldots$$

UpMLs (aka quasi-quotes)

ASTs are the cornerstone of our calculus. But ASTs are verbose. UpMLs (aka quasi-quotes or back-quotes) ameliorate this problem by allowing concrete syntax to be used to represent ASTs.

To add UpMLs to our language, we first extend the grammar as follows:

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

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

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

UpMLs (aka quasi-quotes)

ASTs are the cornerstone of our calculus. But ASTs are verbose. UpMLs (aka quasi-quotes or back-quotes) ameliorate this problem by allowing concrete syntax to be used to represent ASTs.

To add UpMLs to our language, we first extend the grammar as follows:

$$M \quad ::= \quad \ldots \mid \uparrow\{M\} \qquad\qquad t \quad ::= \quad \ldots$$

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

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

Like downMLs, upMLs are disappeared by the compile-time stage.

A subtlety

A subtlety

Recall, we want quasi-quotes, not quotes to be more flexible. I.e. we want ’holes’ in upMLs where we can run arbitrary computation. How can we do that?

A subtlety

Recall, we want quasi-quotes, not quotes to be more flexible. I.e. we want 'holes' in upMLs where we can run arbitrary computation. How can we do that?

A subtlety

Recall, we want quasi-quotes, not quotes to be more flexible. I.e. we want 'holes' in upMLs where we can run arbitrary computation. How can we do that?

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

A subtlety

Recall, we want quasi-quotes, not quotes to be more flexible. I.e. we want 'holes' in upMLs where we can run arbitrary computation. How can we do that?

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

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

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

$$\uparrow\{2+\downarrow\{\uparrow\{3+4\}\}\} \quad \Downarrow_{ct} \quad \mathrm{ast}_{\mathrm{add}}(\mathrm{ast}_{\mathrm{int}}(2), \mathrm{ast}_{\mathrm{add}}(\mathrm{ast}_{\mathrm{int}}(3), \mathrm{ast}_{\mathrm{int}}(4)))$$

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

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

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

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

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

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

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

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).

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 neatly connected as two facets of the same AST-coin.

Other features

Other features

We can easily add other features, like

Other features

We can easily add other features, like

- Lifting, where semi-arbitrary **run-time** values to be lifted up a meta-level, e.g. $\mathsf{lift}(3) \Downarrow_\lambda \mathsf{ast}_{\mathsf{int}}(3)$.

Other features

We can easily add other features, like

- Lifting, where semi-arbitrary **run-time** values to be lifted up a meta-level, e.g. $\text{lift}(3) \Downarrow_\lambda \text{ast}_{\text{int}}(3)$.
- Cross-level variable scoping.

Staged power function $\lambda nx.x^n$

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

$$M = rec\ p.\lambda n.\text{if } n = 1 \text{ then} \uparrow\{x\} \text{ else} \uparrow\{x \times \downarrow\{p\ (n-1)\}\}$$
$$power = \lambda n.\uparrow\{\lambda x.\downarrow\{M\ n\}\}$$

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

$$M = rec\ p.\lambda n.\text{if } n = 1 \text{ then} \uparrow\{x\} \text{ else} \uparrow\{x \times \downarrow\{p\,(n-1)\}\}$$
$$power = \lambda n.\uparrow\{\lambda x.\downarrow\{M\ n\}\}$$

Then *power* 3 reduces to AST equivalent of

$$\uparrow\{\lambda x.x \times x \times x\}$$

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

$$\begin{aligned} M &= rec\ p.\lambda n.\text{if } n = 1 \text{ then } {\uparrow}\{x\} \text{ else } {\uparrow}\{x \times {\downarrow}\{p\ (n-1)\}\} \\ power &= \lambda n.\ {\uparrow}\{\lambda x.\ {\downarrow}\{M\ n\}\} \end{aligned}$$

Then *power* 3 reduces to AST equivalent of

$${\uparrow}\{\lambda x.x \times x \times x\}$$

The function *power* can be used to specialise code at compile-time:

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

$$
\begin{aligned}
M &= rec\ p.\lambda n.\text{if } n = 1 \text{ then} \uparrow\{x\} \text{ else} \uparrow\{x \times \downarrow\{p\ (n-1)\}\} \\
power &= \lambda n. \uparrow\{\lambda x. \downarrow\{M\ n\}\}
\end{aligned}
$$

Then *power* 3 reduces to AST equivalent of

$$\uparrow\{\lambda x.x \times x \times x\}$$

The function *power* can be used to specialise code at compile-time:

$$\text{let } cube\ = \downarrow\{power\ 3\} \text{ in } (cube\ 4) + (cube\ 5)$$

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

$$M = rec\ p.\lambda n.\text{if } n = 1 \text{ then} \uparrow\{x\} \text{ else} \uparrow\{x \times \downarrow\{p\ (n-1)\}\}$$
$$power = \lambda n. \uparrow\{\lambda x. \downarrow\{M\ n\}\}$$

Then *power* 3 reduces to AST equivalent of

$$\uparrow\{\lambda x.x \times x \times x\}$$

The function *power* can be used to specialise code at compile-time:

$$\text{let } cube\ = \downarrow\{power\,3\} \text{ in } (cube\ 4) + (cube\ 5)$$

and at run-time:

Staged power function $\lambda nx.x^n$

We want to specialise $\lambda nx.x^n$ w.r.t. first argument.

$$
\begin{aligned}
M &= rec\ p.\lambda n.\text{if } n = 1 \text{ then} \uparrow\{x\} \text{ else} \uparrow\{x \times \downarrow\{p\ (n-1)\}\} \\
power &= \lambda n. \uparrow\{\lambda x. \downarrow\{M\ n\}\}
\end{aligned}
$$

Then *power* 3 reduces to AST equivalent of

$$\uparrow\{\lambda x.x \times x \times x\}$$

The function *power* can be used to specialise code at compile-time:

$$\text{let } cube \ = \downarrow\{power\, 3\} \text{ in } (cube\, 4) + (cube\, 5)$$

and at run-time:

$$\text{let } cube \ = \ \text{eval}(power\, 3) \text{ in } (cube\, 4) + (cube\, 5)$$

Rational reconstruction?

Rational reconstruction?

We believe that adding HGMP to λ-calculus is simple, yet captures the essence of HGMP.

Rational reconstruction?

We believe that adding HGMP to λ-calculus is simple, yet captures the essence of HGMP.

How do we prove this, when existing approaches to HGMP diverge from our proposals?

Rational reconstruction?

We believe that adding HGMP to λ-calculus is simple, yet captures the essence of HGMP.

How do we prove this, when existing approaches to HGMP diverge from our proposals?

Reflective equilibrium, balance or coherence between model and PL reality.

Rational reconstruction?

We believe that adding HGMP to λ-calculus is simple, yet captures the essence of HGMP.

How do we prove this, when existing approaches to HGMP diverge from our proposals?

Reflective equilibrium, balance or coherence between model and PL reality.

If you have HGMP phenomena that don't agree with our calculus, please contact us.

HGMP$(\cdot)$: mechanical HGMPification of languages

HGMP$(\cdot)$: mechanical HGMPification of languages

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:

$\mathit{HGMP}(\cdot)$

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 appropriate reduction rules for ASTs, upMLs and downMLs with computation driven by the base language. Note that $\mathit{HGMP}(\lambda)$ does not change the reduction rules of λ-calculus itself. **Only adds rules.**

$HGMP(\cdot)$ semi-formally

$HGMP(\cdot)$ semi-formally

Assume C is the set of L's program constructors, L_{mp}'s constructors and tags then:

$$T = C \cup \{\mathrm{eval}, \mathrm{promote}\}$$

$$C_{mp} = C \cup \{\mathrm{eval}, \downarrow\{\_\}, \uparrow\{\_\}\} \cup \{\mathrm{ast}_{\mathrm{t}} \mid t \in T\} \cup \{\mathrm{tag}_{\mathrm{t}} \mid t \in T\}$$

$HGMP(\cdot)$ semi-formally

Assume C is the set of L's program constructors, L_{mp}'s constructors and tags then:

$$T = C \cup \{\mathrm{eval}, \mathrm{promote}\}$$

$$C_{mp} = C \ \cup \ \{\mathrm{eval}, \downarrow\{\_\}, \uparrow\{\_\}\} \cup \{\mathrm{ast_t} \mid t \in T\} \cup \{\mathrm{tag_t} \mid t \in T\}$$

The arities and binders of the new syntax are as follows:

- If $c \in C$ then its arity and binders are unchanged in C_{mp}.
- $\mathrm{ast_c}$ has the same arity as $c \in C$ and no binders.
- $\mathrm{ast_{promote}}$ has variable arity, or, equivalently has arity 2, with the second argument being of type list. There are no binders.
- $\mathrm{ast_{eval}}$ has arity 1 and no binders.
- $\mathrm{tag_t}$ has arity 0 and no binders for $t \in T$.
- eval, $\downarrow\{\_\}$, and $\uparrow\{\_\}$ have arity 1 and no binders.

$HGMP(\cdot)$

$HGMP(\cdot)$

We add the following rules to the operations rules of L (omitting rules for upML for simplicity).

$$\frac{t \in T}{t \Downarrow_\lambda t} \qquad \frac{L \Downarrow_\lambda M \quad M \Downarrow_{dl} N \quad N \Downarrow_\lambda N'}{\mathrm{eval}(L) \Downarrow_\lambda N'}$$

$$\frac{\ldots \quad M_i \Downarrow_\lambda N_i \quad \ldots \quad t \in T}{\mathrm{ast}_{\mathrm{t}}(\tilde{M}) \Downarrow_\lambda \mathrm{ast}_{\mathrm{t}}(\tilde{N})}$$

$HGMP(\cdot)$

We add the following rules to the operations rules of L (omitting rules for upML for simplicity).

$$\frac{t \in T}{t \Downarrow_\lambda t} \qquad \frac{L \Downarrow_\lambda M \quad M \Downarrow_{dl} N \quad N \Downarrow_\lambda N'}{\mathrm{eval}(L) \Downarrow_\lambda N'}$$

$$\frac{\dots \quad M_i \Downarrow_\lambda N_i \quad \dots \quad t \in T}{\mathrm{ast}_{\mathrm{t}}(\tilde{M}) \Downarrow_\lambda \mathrm{ast}_{\mathrm{t}}(\tilde{N})}$$

Constructors with binders are most easily explained by example. If c has arity 2, with the first argument being a binder, the following rule must be added:

$$\frac{M \Downarrow_{dl} \text{''}x\text{''} \quad N \Downarrow_{dl} N'}{\mathrm{ast}_{\mathrm{c}}(M, N) \Downarrow_{dl} c(x, N')}$$

The following rules must be added for higher-order ASTs:

The following rules must be added for higher-order ASTs:

$$\frac{L \Downarrow_{dl} \mathrm{tag}_{\mathrm{t}} \quad M_i \Downarrow_{dl} N_i \quad t \in T}{\mathrm{ast}_{\mathrm{promote}}(L, \tilde{M}) \Downarrow_{dl} \mathrm{ast}_{\mathrm{c}}(\tilde{N})}$$

$$\frac{L \Downarrow_{dl} \mathrm{tag}_{\mathrm{promote}} \quad M \Downarrow_{dl} \mathrm{tag}_{\mathrm{t}} \quad N_i \Downarrow_{dl} R_i}{\mathrm{ast}_{\mathrm{promote}}(L, M, \tilde{N}) \Downarrow_{dl} \mathrm{ast}_{\mathrm{promote}}(\mathrm{tag}_{\mathrm{t}}, \tilde{R})} \qquad \frac{t \in T}{\mathrm{tag}_{\mathrm{t}} \Downarrow_{dl} \mathrm{tag}_{\mathrm{t}}}$$

$HGMP(\cdot)$

Assuming we wish to enable compile-time HGMP, a $\Downarrow_{ct}$ relation must be added:

$$\frac{M \in \{x, ``x"\} \cup \{\mathrm{tag_t} \mid t \in T\}}{M \Downarrow_{ct} M} \qquad \frac{M \Downarrow_{ct} N}{\mathrm{eval}(M) \Downarrow_{ct} \mathrm{eval}(N)}$$

$$\frac{M_i \Downarrow_{ct} N_i \quad c \in C}{c(\tilde{M}) \Downarrow_{ct} c(\tilde{N})} \qquad \frac{M_i \Downarrow_{ct} N_i \quad t \in T}{\mathrm{ast_t}(\tilde{M}) \Downarrow_{ct} \mathrm{ast_t}(\tilde{N})}$$

$$\frac{t \in T}{\mathrm{tag_t} \Downarrow_{ct} \mathrm{tag_t}} \qquad \frac{M \Downarrow_{ct} A \quad A \Downarrow_{\lambda} B \quad B \Downarrow_{dl} C}{\downarrow\{M\} \Downarrow_{ct} C}$$

Questions about $HGMP(\cdot)$

Questions about $HGMP(\cdot)$

- What's a good formal way of specifying source and target languages of $HGMP(\cdot)$?

Questions about $HGMP(\cdot)$

- What's a good formal way of specifying source and target languages of $HGMP(\cdot)$?
- What does it mean for $HGMP(\cdot)$ to be correct?

Questions about $\mathit{HGMP}(\cdot)$

- What's a good formal way of specifying source and target languages of $\mathit{HGMP}(\cdot)$?
- What does it mean for $\mathit{HGMP}(\cdot)$ to be correct?
- Relationship $\mathit{HGMP}(L)$ and $\mathit{HGMP}(\mathit{HGMP}(L))$?

Questions about $\mathit{HGMP}(\cdot)$

- What's a good formal way of specifying source and target languages of $\mathit{HGMP}(\cdot)$?
- What does it mean for $\mathit{HGMP}(\cdot)$ to be correct?
- Relationship $\mathit{HGMP}(L)$ and $\mathit{HGMP}(\mathit{HGMP}(L))$?
- What interesting properties does $\mathit{HGMP}(\cdot)$ preserve?

Questions about $HGMP(\cdot)$

- What's a good formal way of specifying source and target languages of $HGMP(\cdot)$?
- What does it mean for $HGMP(\cdot)$ to be correct?
- Relationship $HGMP(L)$ and $HGMP(HGMP(L))$?
- What interesting properties does $HGMP(\cdot)$ preserve?
- What languages or language features **cannot** be handled satisfactorily by $HGMP(.)$?

Conclusion

Conclusion

- $\mathit{HGMP}(\cdot)$ gives a foundational approach to meta-programming.
- Much work remains.

Conclusion

HGMP$(\cdot)$ gives a foundational approach to meta-programming.
Much work remains.

- Compiler-hooks for *HGMP*$(\cdot)$?

Conclusion

HGMP$(\cdot)$ gives a foundational approach to meta-programming. Much work remains.

- Compiler-hooks for *HGMP*$(\cdot)$?
- Adding hygiene, e.g. using nominal techniques?

Conclusion

HGMP$(\cdot)$ gives a foundational approach to meta-programming. Much work remains.

- Compiler-hooks for *HGMP*$(\cdot)$?
- Adding hygiene, e.g. using nominal techniques?
- Extension of *HGMP*$(\cdot)$ to typed base-languages, using staged typing a la Template Haskell?

Conclusion

HGMP$(\cdot)$ gives a foundational approach to meta-programming. Much work remains.

- Compiler-hooks for *HGMP*$(\cdot)$?
- Adding hygiene, e.g. using nominal techniques?
- Extension of *HGMP*$(\cdot)$ to typed base-languages, using staged typing a la Template Haskell?
- Implementation of *HGMP*$(\cdot)$ and applying it to real languages?

Conclusion

HGMP$(\cdot)$ gives a foundational approach to meta-programming. Much work remains.

- Compiler-hooks for *HGMP*$(\cdot)$?
- Adding hygiene, e.g. using nominal techniques?
- Extension of *HGMP*$(\cdot)$ to typed base-languages, using staged typing a la Template Haskell?
- Implementation of *HGMP*$(\cdot)$ and applying it to real languages?
- Application to proof assistants, e.g. "Engineering Proof by Reflection in Agda" by Swierstra et al to implement tactics?

Conclusion

HGMP($\cdot$) gives a foundational approach to meta-programming. Much work remains.

- Compiler-hooks for *HGMP*($\cdot$)?
- Adding hygiene, e.g. using nominal techniques?
- Extension of *HGMP*($\cdot$) to typed base-languages, using staged typing a la Template Haskell?
- Implementation of *HGMP*($\cdot$) and applying it to real languages?
- Application to proof assistants, e.g. “Engineering Proof by Reflection in Agda” by Swierstra et al to implement tactics?
- Hoare logics and other specification mechanism. Can *HGMP*($\cdot$) be extended to transform logic for L to logic for *HGMP*(L)?

Conclusion

$\mathit{HGMP}(\cdot)$ gives a foundational approach to meta-programming. Much work remains.

- Compiler-hooks for $\mathit{HGMP}(\cdot)$?
- Adding hygiene, e.g. using nominal techniques?
- Extension of $\mathit{HGMP}(\cdot)$ to typed base-languages, using staged typing a la Template Haskell?
- Implementation of $\mathit{HGMP}(\cdot)$ and applying it to real languages?
- Application to proof assistants, e.g. "Engineering Proof by Reflection in Agda" by Swierstra et al to implement tactics?
- Hoare logics and other specification mechanism. Can $\mathit{HGMP}(\cdot)$ be extended to transform logic for L to logic for $\mathit{HGMP}(L)$?
- Generalising $\mathit{HGMP}(\cdot)$ to heterogeneous meta-programming?

Questions?