A Multi-stage Monadic Metalanguage

Eugenio Moggi
moggi@disi.unige.it

DISI, Univ. of Genova
(joint work with S.Fagorzi)
Overview: Motivations

Why MSP (Multi-stage programming)?

1) Staging (Partial Evaluation) is an optimization technique usable in many areas: exploit info at early stages to generate customized code for later stages.

2) GC (Global Computing) applications operate with incomplete information about a dynamically changing environment, thus they need to interleave
   - meta-programming activities (code generation, assembly of components)
   - normal computational activities (IO, modification of local store, . . .)
   - security checks (at administrative domain boundaries)

MSP supports interleaving of code generation and normal computation.
Overview: Aims

- MMML (Multi-stage Monadic MetaLanguage) provides reference semantics describing the subtle interactions between staging (code generation) and computational effects.
- MMML adopts simple type system (fails to statically detect all run-time errors):
  - no best option for a static type system [TD99, CMS??, Nan02, NT03],
  - there are cases when dynamic error detection is preferable [FMP02].
Overview: Aims and Approach

- MMML (Multi-stage Monadic MetaLanguage) provides reference semantics describing the subtle interactions between staging (code generation) and computational effects.
- MMML adopts simple type system (fails to statically detect all run-time errors):
  - no best option for a static type system [TD99, CMS??, Nan02, NT03],
  - there are cases when dynamic error detection is preferable [FMP02].

- computational types $M^\tau$: explicit distinction between values and computations
- code type $\langle \tau \rangle$ (as in MetaML): reflective nature of languages for MSP
- annotated term constructors: operationally relevant information explicit in terms (as in multi-level languages for binding-time analysis)
- terms à la CRS (Combinatory Reduction Systems): for syntactic uniformity
- ChAM-like (Chemical Abstract Machine) approach:
  - transparent simplification $\longrightarrow$ (like ChAM heating) – PURITY
  - programmable computation $\longleftarrow$ (like ChAM reduction)
Technical Part on MMML

Article stresses also that the addition of staging to pre-existing monadic metalanguage is MODULAR!
MMML: Syntax

\[ \tau \in T : = X \mid M\tau \mid \text{ref } \tau \mid \tau_1 \rightarrow \tau_2 \mid \langle \tau \rangle \text{ type} \]

\[ e \in E : = x \mid l \mid c([x_i]e_i \mid i \in m) \text{ term, } x \in X \text{ variable, } l \in L \text{ location} \]

\[ c \in C : = \text{ret} \mid \text{do} \mid \text{new} \mid \text{get} \mid \text{set} \mid \lambda \mid @ \mid \text{up} \mid \text{dn} \mid c_V \mid c_M \text{ term constructor} \]

Syntax of terms à la CRS [Klo80]

- \( \lambda \)-abstraction \( \lambda x.e \) becomes \( \lambda([x]e) \)
- application \( e_1 e_2 \) becomes \( @\langle e_1, e_2 \rangle \)
- Haskell’s do notation \( \text{do}\{x \leftarrow e_1; e_2\} \) becomes \( \text{do}(e_1, [x]e_2) \)

Alternative BNF for \( c \in C \)

\[ op \in Op : = \text{ret} \mid \text{do} \mid \text{new} \mid \text{get} \mid \text{set} \mid \lambda \mid @ \mid \text{up} \mid \text{dn} \text{ ordinary term constructor} \]

\[ c \in C : = op_B \text{ annotated term constructor } B \in \{V, M\}^* \]

Warning: in multi-level languages (e.g. in \( \lambda^M \) of [GJ95, Dav96]) \( c_V \) and \( c_M \) are identified, i.e. \( op_B \) becomes \( op_n \) where \( n \) is the length of the sequence \( B \).
MMML: Naive Type System

- $\tau \in \mathcal{T} ::= X \mid M\tau \mid \text{ref } \tau \mid \tau_1 \rightarrow \tau_2 \mid \langle \tau \rangle$ type

- $e \in \mathcal{E} ::= x \mid l \mid c([\overline{x}_i]e_i \mid i \in m)$ term, $x \in X \text{ variable}, \; l \in L \text{ location}$

- $c \in \mathcal{C} ::= \text{ret} \mid \text{do} \mid \text{new} \mid \text{get} \mid \text{set} \mid \lambda \mid @ \mid \text{up} \mid \text{dn} \mid c_V \mid c_M \text{ term constructor}$

- **standard** $\Gamma \vdash \Sigma \; e: \tau$ typing judgment, where $\Gamma$ type assignment $x: \tau$ to variables

\[
\begin{align*}
x & \quad \frac{\varepsilon}{\Gamma \vdash \Sigma \; x: \tau} \\
& \quad \frac{\varepsilon}{\Gamma \vdash \Sigma \; x: \tau} \\
\end{align*}
\]

\[
\begin{align*}
x & \quad \frac{\varepsilon}{\Gamma \vdash \Sigma \; x: \tau} \\
\end{align*}
\]

- each term constructor $c$ has a type schema $\langle \overline{\tau_i} \Rightarrow \tau_i \mid i \in m \rangle \Rightarrow \tau$
MMML: Type Schema for term constructors

- $\tau \in T ::= X \mid M\tau \mid \text{ref } \tau \mid \tau_1 \rightarrow \tau_2 \mid \langle \tau \rangle$ type

- $e \in E ::= x \mid l \mid c([\overline{x}_i]e_i \mid i \in m)$ term, $x \in X$ variable, $l \in L$ location

- $c \in C ::= \text{ret} \mid \text{do} \mid \text{new} \mid \text{get} \mid \text{set} \mid \lambda \mid \@ \mid \text{up} \mid \text{dn} \mid c_V \mid c_M$ term constructor

- $\text{ret}: \tau \Rightarrow M\tau$ return value, $\text{do}: M\tau_1, (\tau_1 \Rightarrow M\tau_2) \Rightarrow M\tau_2$ sequential execution

- $\text{new}: \tau \Rightarrow M(\text{ref } \tau)$, $\text{get}: \text{ref } \tau \Rightarrow M\tau$, $\text{set}: \text{ref } \tau, \tau \Rightarrow M(\text{ref } \tau)$ store operations

- $\lambda: (\tau_1 \Rightarrow \tau_2) \Rightarrow (\tau_1 \rightarrow \tau_2)$ abstraction, $\@: (\tau_1 \rightarrow \tau_2), \tau_1 \Rightarrow \tau_2$ application

- $\text{up}: \tau \Rightarrow \langle \tau \rangle$ cross stage persistence, $\text{dn}: \langle \tau \rangle \Rightarrow M\tau$ compilation

- If $c: (\overline{\tau}_i \Rightarrow \tau_i \mid i \in m) \Rightarrow \tau$, then
  - $c_V: (\langle \overline{\tau}_i \rangle \Rightarrow \langle \tau_i \rangle \mid i \in m) \Rightarrow \langle \tau \rangle$ code value constructor
  - $c_M: (\langle \overline{\tau}_i \rangle \Rightarrow M\langle \tau_i \rangle \mid i \in m) \Rightarrow M\langle \tau \rangle$ code generation

where $\langle \tau_i \mid i \in m \rangle$ stands for $(\langle \tau_i \rangle \mid i \in m)$. For instance

- $\lambda_V: (\langle \tau_1 \rangle \Rightarrow \langle \tau_2 \rangle) \Rightarrow \langle \tau_1 \rightarrow \tau_2 \rangle$ code for abstraction

- $\lambda_M: (\langle \tau_1 \rangle \Rightarrow M\langle \tau_2 \rangle) \Rightarrow M\langle \tau_1 \rightarrow \tau_2 \rangle$ generate code for abstraction
MMML: Type Schema for term constructors

- $\tau \in T ::= X \mid M\tau \mid \text{ref } \tau \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle$ type

- $e \in E ::= x \mid l \mid c([\overline{x_i}] e_i \mid i \in m)$ term, $x \in X$ variable, $l \in L$ location

- $c \in C ::= \text{ret} \mid \text{do} \mid \text{new} \mid \text{get} \mid \text{set} \mid \lambda \mid @ \mid \text{up} \mid \text{dn} \mid c_V \mid c_M$ term constructor

- $\text{up}: \tau \Rightarrow \langle \tau \rangle$ cross stage persistence (binary inclusion)

- $\text{dn}: \langle \tau \rangle \Rightarrow M\tau$ compilation (may raise unresolved link error)

- If $\lambda:(\tau_1 \Rightarrow \tau_2) \Rightarrow (\tau_1 \to \tau_2)$, then
  - $\lambda_V: (\langle \tau_1 \rangle \Rightarrow \langle \tau_2 \rangle) \Rightarrow \langle \tau_1 \to \tau_2 \rangle$ code for abstraction
  - $\lambda_M: (\langle \tau_1 \rangle \Rightarrow M\langle \tau_2 \rangle) \Rightarrow M\langle \tau_1 \to \tau_2 \rangle$ generate code for abstraction
Operational Semantics: ChAM-like approach

- $Id \rightarrow Id'$ simplification is $\beta$-reduction $\odot(\lambda([x]e_2), e_1) \rightarrow e_2[x := e_1]$

- $Id \cross Id' | done | err$ computation, where

- $Id \equiv (X|\mu, e, E) \in$ Conf program configuration (closed system)
  - $X \in \mathcal{P}_{fin}(X)$ set of names generated so far
  - $\mu \in S \triangleq L \xrightarrow{fin} E$ current store
  - $e$ program fragment under consideration
  - $E \in EC: :: = \square | do(E, e) | c_M (v, [x]E, f)$ evaluation context for $e$
    where $f:: = [x]e$ abstraction and $v:: = [x]ret(e)$ value abstraction
Operational Semantics: Properties

- $Id \longrightarrow Id'$ simplification is $\beta$-reduction $\Rightarrow (\lambda([x] e_2), e_1) \longrightarrow e_2[x := e_1]$
- $Id \vdash Id'$ | done | err computation, where
- $Id \equiv (X | \mu, e, E) \in \text{Conf program configuration (closed system)}$

CR $\longrightarrow$ is confluent, and induced equivalence $\equiv$ is a congruence

$$
Id_1 \longrightarrow D_1
\quad \downarrow
\quad \therefore

Id_2 \vdash \longrightarrow D_2
$$

computation is preserved by further simplification, i.e.

SR (Subject Reduction) $Id$ well-formed and $Id \longrightarrow Id'$ implies $Id'$ well-formed

PR (Progress) $Id$ well-formed implies $Id \longrightarrow$, where $\longrightarrow \triangleq \quad \cup$
Operational Semantics: Computation Rules I

- \( Id \rightarrow Id' \) simplification is \( \beta \)-reduction \( \tau(\lambda([x]e_2), e_1) \rightarrow e_2[x := e_1] \)
- \( Id \trianglelefteq Id' \mid \) done \mid err computation, where
- \( Id \equiv (X|\mu, e, E) \in \text{Conf} \) program configuration (closed system)

**Administrative Steps**

A.0 \( (X|\mu, \text{ret}(e), \Box) \trianglelefteq \) done

A.1 \( (X|\mu, \text{do}(e_1, [x]e_2), E) \trianglelefteq (X|\mu, e_1, E[\text{do}(\Box, [x]e_2)]) \)

A.2 \( (X|\mu, \text{ret}(e_1), E[\text{do}(\Box, [x]e_2)]) \trianglelefteq (X|\mu, e_2[x := e_1], E) \)

**Imperative Steps**

I.1 \( (X|\mu, \text{new}(e), E) \trianglelefteq (X|\mu\{l := e\}, \text{ret}(l), E) \) where \( l \notin \text{dom}(\mu) \)

I.2 \( (X|\mu, \text{get}(l), E) \trianglelefteq (X|\mu, \text{ret}(e), E) \) provided \( e = \mu(l) \)

I.3 \( (X|\mu, \text{set}(l, e), E) \trianglelefteq (X|\mu\{l := e\}, \text{ret}(l), E) \) provided \( l \in \text{dom}(\mu) \)
Operational Semantics: Computation Rules II

- \( Id \longrightarrow Id' \) simplification is \( \beta \)-reduction

\[ @ (\lambda([x]e_2), e_1) \longrightarrow e_2[x := e_1] \]

- \( Id \longmapsto Id' \mid \text{done} \mid \text{err} \) computation, where

- \( Id \equiv (X|\mu, e, E) \in \text{Conf program configuration (closed system)} \)

Code Generation Steps (only unary case \( \lambda_M \))

- **G.1** \( (X|\mu, \lambda_M([x]e), E) \longmapsto (X, x|\mu, e, E[\lambda_M([x]\Box)]) \)
  
  rename \( x \) to avoid clashes with \( X \)

- **G.2** \( (X|\mu, \text{ret}(e), E[\lambda_M([x]\Box)]) \longmapsto (X|\mu, \text{ret}(\lambda_V([x]e)), E) \)
  
  occurrences of \( x \) in \( e \) are \text{re-captured}
Operational Semantics: Computation Rules II

- $Id \longrightarrow Id'$ simplification is $\beta$-reduction @ $(\lambda([x]e_2), e_1) \longrightarrow e_2[x := e_1]$
- $Id \longmapsto Id'$ | done | err computation, where
- $Id \equiv (X|\mu, e, E) \in \text{Conf program configuration (closed system)}$

Compilation and Run-time Errors

C.1 $\quad (X|\mu, \text{dn}(vc), E) \longmapsto \left\{ \begin{array}{l}
(X|\mu, \text{ret}(e), E) \quad \text{if } e = vc \downarrow \\
\text{err} \quad \text{if } vc \downarrow \text{undefined}
\end{array} \right.$

where $vc \in VC: ::= x \mid \text{up}(e) \mid c_V([\overline{x}_i]vc_i | i \in m)$ code value and

Demotion $vc \downarrow$ maps code value $vc: \langle \tau \rangle$ to represented term $e: \tau$

- $x \downarrow \text{undefined}$
- $\text{up}(e) \downarrow = e$ (base case, like $x$)
- $c_V([\overline{x}_i]vc_i | i \in m) \downarrow = c([\overline{x}_i]e_i | i \in m)$

provided $e_i = vc_i[\overline{x}_i := \text{up}(\overline{x}_i)] \downarrow$ for $i \in m$
An important Lemma

If $(X | \mu, e, E) \longrightarrow (X' | \mu', e', E')$ and

- $\text{FV}(\mu, e) \cup \text{CV}(E) \subseteq X$ and $\text{FV}(E) \subseteq X - \text{CV}(E)$

then $X \subseteq X'$, $\text{dom}(\mu) \subseteq \text{dom}(\mu')$ and

- $\text{FV}(\mu', e') \cup \text{CV}(E') \subseteq X'$ and $\text{FV}(E') \subseteq X' - \text{CV}(E')$

where $\text{FV}(\_)$ set of free variables, $\text{CV}(\_)$ set of captured variables, e.g.

- $\text{FV}(c_M(\overline{v}, [x]E, \overline{f})) \triangleq \text{FV}(\overline{v}, \overline{f}) \cup (\text{FV}(E) - \overline{x})$

- $\text{CV}(c_M(\overline{v}, [x]E, \overline{f})) \triangleq \overline{x} \cup \text{CV}(E)$
An important Lemma

If \((X|\mu, e, E) \longrightarrow (X'|\mu', e', E')\) and

- \(\text{FV}(\mu, e) \cup \text{CV}(E) \subseteq X\) and \(\text{FV}(E) \subseteq X - \text{CV}(E)\)

then \(X \subseteq X'\), \(\text{dom}(\mu) \subseteq \text{dom}(\mu')\) and

- \(\text{FV}(\mu', e') \cup \text{CV}(E') \subseteq X'\) and \(\text{FV}(E') \subseteq X' - \text{CV}(E')\)

Well-formed Configuration \(\Delta \vdash_{\Sigma} (X|\mu, e, E) \iff \Delta\)

dom(\Sigma) = dom(\mu) and \(\text{dom}(\Delta) = X\) [\(\Delta\) assigns code types to \(x \in X\)] and

- \(\mu(l) = e_l\) and \(\Sigma(l) = \tau_l\) imply \(\Delta \vdash_{\Sigma} e_l: \tau_l\)

- exists \(\tau\) s.t. \(\Delta \vdash_{\Sigma} e: M\tau\)

- exists \(\tau'\) s.t. \(\Delta, \Box: M\tau \vdash_{\Sigma} E: M\tau'\)

\[
\lambda_M \quad \frac{\Delta, \Box: M\langle\tau_1 \rightarrow \tau_2\rangle \vdash_{\Sigma} E: M\tau'}{
\Delta, x: \langle\tau_1\rangle, \Box: M\langle\tau_2\rangle \vdash_{\Sigma} E[\lambda_M([x]\Box)]: M\tau'}
\]

by the lemma \(x\) should not occur in \(E\).
Summary of Achievements

We have introduced a multi-stage monadic metalanguage MMML, main advantages

- SEPARATE simplification (no user control on simplification strategy) and computation (user control on order of evaluation)
- ALL operationally relevant information explicitly in terms

smooth extension to multi-staging:
  - NO CHANGE to simplification,
  - only ONE new clause for evaluation contexts,
  - and FEW additional computation rules

- BETTER handling of scope extrusion, by keeping a log of generated names
Further Work on Monadic Metalanguages

- simplification: one may replace $\lambda$-calculus with other pure calculi
- computation: one may consider multi-lingual extensions, where a variety of programming languages (monads) coexist
  - multiple monads, but only one code type constructor
  - one $c_{M_i}$ for each monad, but
  - only one code value constructor $c_V$

$e: M_1\langle M_2\tau \rangle$ program in $PL_1$ generating representation of program in $PL_2$
Further Work on Compilation

May compilation become a simplification rule with refined typing?

- $dn: [\langle \tau \rangle] \Rightarrow [\tau]$ exploit close types [CMS??]
- $dn: \Box (\tau [\emptyset]) \Rightarrow \tau$ similar idea using refined code types [Nan02]
- $dn: (\forall \alpha. \langle \tau \rangle^\alpha) \Rightarrow \tau$ exploit encapsulation of representation [NT03]

main difficulty: proof of SR for simplification.

Alternative compilation rules, e.g. use lazy demotion (some clauses):

- $up_{\nu_n}(e) \Downarrow = e$
- $ret_{\nu_{n+1}}(e) \Downarrow = dn_{\nu_n}(e)$
Further Work on Implementation Issues

The ChAM-like semantics for MMML is too abstract
- terms are considered modulo α-conversion
- simplification is β-reduction (instead of a deterministic strategy)
- substitution does automatic renaming of bound variables

Implementations should be consistent with the reference semantics, but
- manipulate concrete terms
- use a naive substitution
- adopt a deterministic simplification strategy
- make explicit generation of fresh names (gensym)