

## MetaML: structural reflection for multi-stage programming (CalcagnoC, MoggiE, in collaboration with TahaW)

- starting point: functional languages, MetaML [MetaML]
- keywords: programs as data, partial eval., code generation, staging
- aim: incorporate multi-stage programming constructs in existing languages
- related topics: separate compilation and linking [Car97@POPL]

### Work done [CMT00@ICALP, CM00@SAIG]

- **safe combination** of imperative and multi-stage programming  
ref  $c$  well-formed only when  $c$  **closed type**
- MetaML **conservative extension** of ML w.r.t. typing  $e:t$  and  
run-time observations: termination  $e \Downarrow$ , run-time error  $e \Downarrow \text{err}$   
**Conjecture**: conservative ext. also w.r.t. observational equiv.

**Remark.** Results should extend in presence of other computational effects.

# Terminology

**Staged computation:** explicit division of a computation in stages; staging is an **intensional** property, concerned with how values are computed.

**Partial evaluation:** (static) specialization of a program based on partial data. Multi-level languages, **open** code,  $\lambda^\circ$  [Dav96@LICS]

**Run-time code generation:** dynamic generation of portions of program. **Closed** code,  $\lambda^\square$  [DP96@POPL]

## MetaML design principles

- **explicit:** power of staging in the hands of programmer
- **reflective:** source code represented as data
- **static:** staging errors should be type errors

## Summary of Talk

- informal description of multi-stage programming constructs
- sample applications:
  - 1) generative components
  - 2) linking of code to services
  - 3) mobile code abstraction in **MetaKlaim**
- formal developments: syntax, type system, operational semantics

# MetaML types and constructs for multi-stage programming

Code type  $\langle t \rangle$ :  $\frac{\Gamma \vdash_{obj} t}{\Gamma \vdash_{meta} \langle t \rangle}$  obj-level type  
meta-level type

Brackets  $\langle e \rangle$ :  $\frac{\Gamma \vdash_{obj} e:t}{\Gamma \vdash_{meta} \langle e \rangle: \langle t \rangle}$  obj-level **program fragment** of type  $t$   
meta-level **representation** of  $e$

Escape  $\sim e$ :  $\frac{\Gamma \vdash_{meta} e: \langle t \rangle}{\Gamma \vdash_{obj} \sim e: t}$   $\sim \langle e \rangle \rightarrow e$

cross-stage persistence  $\%e$ :  $\frac{\Gamma \vdash_{meta} e: t}{\Gamma \vdash_{obj} \%e: t}$  **meta-level type  $t$  must be an obj-level type**

Closed type  $[t]$ :  $\frac{\Gamma \vdash_{meta} t}{\Gamma \vdash_{meta} [t]}$  meta-level type  
subset of  $e:t$  without **unresolved links**

Run/execution  $\text{run } e$ :  $\frac{\Gamma \vdash_{meta} e: [\langle t \rangle]}{\Gamma \vdash_{meta} \text{run } e: t}$  **obj-level type  $t$  must be a meta-level type**  $\text{run } \langle \%e \rangle \rightarrow e$

**Remark.** In MetaML obj-level = meta-level: **reflective tower**.

## What is a representation $\langle e \rangle$ of a program fragment?

- **observable** source code: allows code analysis, requires open source
- **intermediate code**: support portability
- **non-observable** binary code: compatible with proprietary SW components

### Cross-stage persistence $\%e$

Allows to include **native code** for  $e$  in obj-level programs. Code analysis inside  $\%e$  is not allowed, since  $e$  is not at obj-level.

### Code execution $\text{run } e$ and closed types

$e$  should evaluate to  $\langle e' \rangle$ , where  $e'$  is an obj-level program fragment,  $e'$  can be safely executed only if it has no **unresolved links** (complete program).

## Lightweight and Generative components [KCC99@GCSE,KCC00@SAIG]

- Lightweight: when function call inefficient or semantically inappropriate
- Generative: embodies method for constructing code

Component = **higher-order macro** written in functional meta-language to generate code in an **imperative** object language.

### Examples of components [KCC99@GCSE]

- **sorting**  $\text{nat} \rightarrow [(\langle X \rangle \rightarrow \langle X \rangle \rightarrow \langle \text{bool} \rangle) \rightarrow \langle \text{array } X \rangle \rightarrow \langle \text{unit} \rangle]$   
parameters: size of array, code for comparison
- **caching**  $\text{nat} \rightarrow X \rightarrow [(\langle \text{nat} \rightarrow X \rangle \rightarrow \langle \text{nat} \rangle \rightarrow \langle X \rangle) \rightarrow \langle \text{nat} \rightarrow X \rangle]$   
parameters: size of cache, dummy value, code (open recursion) of function
- vector operations: loop-fusion, no arrays for intermediate results

## Parameterized procedure *Sort*

*Sort*:  $\text{nat} \rightarrow (X \rightarrow X \rightarrow \text{bool}) \rightarrow \text{array } X \rightarrow \text{unit}$  sort procedure with parameters

- $n$ :  $\text{nat}$  size of array  $A$  to sort
- $leq$ :  $X \rightarrow X \rightarrow \text{bool}$  comparison function

## Generative component *SortComp* [KCC00@SAIG]

*SortComp*:  $\text{nat} \rightarrow [(\langle X \rangle \rightarrow \langle X \rangle \rightarrow \langle \text{bool} \rangle) \rightarrow \langle \text{array } X \rangle \rightarrow \langle \text{unit} \rangle]$  with parameters

- $n$ :  $\text{nat}$  **statically known** size of array to sort
- $leq'$ :  $\langle X \rangle \rightarrow \langle X \rangle \rightarrow \langle \text{bool} \rangle$  **generic** code for comparing two elements

## Advantages of *SortComp* over *Sort*

- component can generate optimized code for size  $n$
- client can give optimized comparison code instead of calling  $leq$   
 $leq' \ x \ y = \langle \%leq \ \tilde{x} \ \tilde{y} \rangle$
- client can inline generated code instead of wrapping it in a procedure  
 $\langle \text{fn } A \Rightarrow \sim(\text{SortComp } n \ leq' \ \langle A \rangle) \rangle$

## Example: linking of generic code to services

**Generic Code**  $gc: [(Serv \rightarrow \langle In \rangle \rightarrow \langle Out \rangle) \rightarrow \langle Cmd \rangle]$

```
gc call = < var x1,x2;  
           x1 := ~(call s1 <i>);  
           if "test" then x2 := ~(call s2 <x1>)  
                           else x2 := ~(call s1 <x1>); >
```

**Link-Time:** link services (to generic code) before code execution.

```
-| array req = (false|s:Serv) : bool array; (* requested services *)  
-| array srv = (undef|s:Serv) : (In -> Out) array; (* stubs *)  
-| fun call s x = if not req[s] then "update req[s] and srv[s]";  
                  <%srv[s] ~x>;  
val call = ... : Serv -> <In> -> <Out>  
-| val sc = gc call; (* specialize gc and open requested services *)  
val sc : <Cmd>  
-| run sc; (* execute code sc *)  
-| for s:Serv do if req[s] then "close service s";
```

**Run-Time** alternative: link services during code execution.

## Dynamic Newsgatherer: autonomous agent

```
ma(i:Item,u:Loc) = (* Mobile Agent moves and acts autonomously *)
  read(i,!x:Data)@self. out(x)@u. nil
| read(i,!u':Loc)@self. eval(ma(i,u))@u'. nil
```

## Dynamic Newsgatherer: agent abstraction

```
maa(i:Item,u:Loc) = (* Mobile Agent Abstraction needs env to act *)
  fn env:Env => with env do
    read(i,!x:Data)@self. out(x)@u. nil
  | read(i,!u':Loc)@self. out(maa(i,u),"from",u)@u'. nil
```

```
execute = (* process executing maa with suitable env *)
  in(!X:Env=>Proc,"from",!u:Loc)@self.
  (X "env for u" | execute)
```

## Dynamic Newsgatherer: code abstraction

`mca(i:Item,u:Loc) = (* Mobile Code Abstraction needs env' to run *)`

`fn env':Env' ⇒ with env' do`

`< read'(%i,!x:Data)@self'. out'(x)@%u. nil'`

`| read'(%i,!u':Loc)@self'. out'%(mca(i,u)," from" ,u)@u'. nil' >`

when `op:⟨ti⟩⇒⟨t⟩ op'(ei)` stands for `~op(⟨ei⟩)`.

`execute' = (* process running mca with suitable env' *)`

`in(!X:[Env'⇒ <Proc>], " from" ,!u:Loc)@self.`

`(run(X "env' for u") | execute')`

## Related Work

**First-class modules** (AnconaD, Lagorio, Zucca). Parameter `env/env'` for agent/code abstraction are `modules = collection of declarations`.

**Reflective middleware** (Cazzola, AnconaM). Separation between application concerns and communication issues, e.g. synchronization, security.

## Color Conventions for Newsgatherer Example

Things to stress

Type information

`op` Klaim operation, or component of record `env`

`op'` obj-level operation corresponding to generative component of record `env'`

# MetaML types, closed types and terms [CM00@SAIG]

$t \in T ::= \text{nat} \mid t_1 \rightarrow t_2 \mid \text{ref } c \mid [t] \mid \langle t \rangle$  type

$c \in C ::= \text{nat} \mid t_1 \rightarrow c_2 \mid \text{ref } c \mid [t]$  closed type

type equality  $[c] = c$  and kind inclusion  $C \subset T$ .

$e \in E ::= \dots \mid x \mid \lambda x.e \mid e_1 e_2 \mid \text{ref } e \mid !e \mid e_1 := e_2 \mid l \mid$   
 $\langle e \rangle \mid \tilde{e} \mid \%e \mid \text{run } e \mid \text{let } x = e_1 \text{ in } e_2 \mid \bullet e \mid (x)e$

$x \in X$  variable,  $l \in L$  location.

**Type system**  $\Sigma; \Delta; \Gamma \vdash_n e : t$  “ $e$  has type  $t$  and level  $n$ ”

- $\Sigma : L \xrightarrow{fin} C$  signature
- $\Delta : X \xrightarrow{fin} (C \times N)$  **closed** type-and-level assignment
- $\Gamma : X \xrightarrow{fin} (T \times N)$  open type-and-level assignment

$$\frac{\begin{array}{l} \Sigma; \Delta^{\leq m}; \emptyset \vdash_m e_1 : c_1 \\ \Sigma; \Delta, x : c_1^m; \Gamma \vdash_n e_2 : t_2 \end{array}}{\Sigma; \Delta; \Gamma \vdash_n e_2[x := e_1] : t_2}$$

$$\frac{\begin{array}{l} \Sigma; \Delta; \Gamma \vdash_m e_1 : t_1 \\ \Sigma; \Delta; \Gamma, x : t_1^m \vdash_n e_2 : t_2 \end{array}}{\Sigma; \Delta; \Gamma \vdash_n e_2[x := e_1] : t_2}$$

## Why $e: \langle t \rangle$ is not safe for `run e`: unresolved links

```
-| val f = <fn x => ~(run <x>; <0>)>; (* cannot execute x *)
```

## Why `ref <t>` is not a type: scope extrusion

```
-| val l = ref <0>;  
val l = ... : ref <nat>  
-| val f = <fn x => ~(l:=<x>; <0>)>;  
val f = <fn x => 0> : <nat -> nat>  
-| val c = !l;  
val c = <x> : <nat> (* x out of scope *)
```

## Why $\bullet e$ and $(x)e$ needed: marking of useless variables

```
-| val l = ref(fn x => 0);  
val l = ... : (nat -> nat) ref  
-| val f = <fn x => ~(l:=(fn y =>((fn z => 0) <x>)); <0>)>;  
val f = <fn x => 0> : <nat -> nat>
```

x gets out of scope, but it is useless!

## Selection of typing rules

$$\begin{array}{c}
 \frac{}{\Sigma; \Delta; \Gamma \vdash_n x : t} \quad (\Delta, \Gamma)(x) = t^n \quad \frac{\Sigma; \Delta; \Gamma, x : t_1^n \vdash_n e : t_2}{\Sigma; \Delta; \Gamma \vdash_n \lambda x. e : t_1 \rightarrow t_2} \\
 \\
 \frac{\Sigma; \Delta; \Gamma \vdash_{n+1} e : t}{\Sigma; \Delta; \Gamma \vdash_n \langle e \rangle : \langle t \rangle} \quad \frac{\Sigma; \Delta; \Gamma \vdash_n e : \langle t \rangle}{\Sigma; \Delta; \Gamma \vdash_{n+1} \tilde{e} : t} \quad \frac{\Sigma; \Delta; \Gamma \vdash_n e : t}{\Sigma; \Delta; \Gamma \vdash_{n+1} \%e : t} \\
 \\
 \frac{\Sigma; \Delta; \Gamma \vdash_n e : [\langle t \rangle]}{\Sigma; \Delta; \Gamma \vdash_n \text{run } e : t} \quad \frac{\Sigma; \Delta^{\leq n}; \emptyset \vdash_n e : t}{\Sigma; \Delta; \Gamma \vdash_n e : [t]} \quad \frac{\Sigma; \Delta; \Gamma \vdash_n e : [t]}{\Sigma; \Delta; \Gamma \vdash_n e : t} \\
 \\
 \frac{\Sigma; \Delta; \Gamma \vdash_n e_1 : c \quad \Sigma; \Delta, x : c^n; \Gamma \vdash_n e_2 : t}{\Sigma; \Delta; \Gamma \vdash_n \text{let } x = e_1 \text{ in } e_2 : t} \quad \frac{\Sigma; \Delta_1^{>n}; \Gamma_1^{>n} \vdash_n e : c}{\Sigma; \Delta; \Gamma \vdash_n \bullet e : c} \\
 \\
 \frac{\Sigma; \Delta; \Gamma, x : t_1^m \vdash_n e : c}{\Sigma; \Delta; \Gamma \vdash_n (x)e : c} \quad m > n \quad \frac{\Sigma; \Delta, x : c_1^m; \Gamma \vdash_n e : c}{\Sigma; \Delta; \Gamma \vdash_n (x)e : c} \quad m > n
 \end{array}$$

## Binders for marking useless variables

- Bullet  $\bullet e$  binds all free variables in  $e$
- Bind  $(x)e$  binds  $x$  in  $e$

## Typing rules for type equality

$$\frac{\Sigma; \Delta; \Gamma \vdash_n e: t_1}{\Sigma; \Delta; \Gamma \vdash_n e: t_2} t_1 = t_2 \quad \text{recall that } [c] = c$$

## Let-binder in MetaML

let  $x = e_1$  in  $e_2$  not equivalent to  $(\lambda x. e_2) e_1$

- $e_1$  must have a closed type  $c$
- $x: c^n$  is added to  $\Delta$  instead of  $\Gamma$

# CBV operational semantics

$\mu, e \xrightarrow{n} d$  evaluation at level  $n$ , where  $d \equiv \mu', v^n \mid \text{err}$

-  $v^n \in V^n$  value at level  $n$

$v^0 \in V^0 ::= \dots \mid \bullet \lambda x.e \mid \bullet \langle v^1 \rangle \mid \bullet l \mid \lambda x.e \mid \langle v^1 \rangle$

$v^1 \in V^1 ::= \dots \mid x \mid \lambda x.v^1 \mid v_1^1 v_2^1 \mid \dots \mid l \mid$

$\langle v^2 \rangle \mid \tilde{v}^0 \mid \%e \mid \text{run } v^1 \mid \text{let } x = v_1^1 \text{ in } v_2^1 \mid \bullet v^1 \mid (x)v^1$

-  $\mu, \mu': L \xrightarrow{\text{fin}} V_0^0$  value store

## Technical Results

**Closedness.**  $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 \langle v^1 \rangle: [\langle t \rangle]$  implies  $\text{FV}(v^1) = \emptyset$

**Demotion.**  $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_{n+1} v^{n+1}: t$  implies  $\Sigma; \Delta; \Gamma \vdash_n v^{n+1} \downarrow: t$

**Type Safety.**  $\mu, e \xrightarrow{n} d$  and  $\Sigma \models \mu$  and  $\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash_n e: t$  imply exist  $\mu'$  and  $v^n$  and  $\Sigma'$  such that  $d \equiv (\mu', v^n)$  and  $\Sigma, \Sigma' \models \mu'$  and  $\Sigma, \Sigma', \Delta^{+1}; \Gamma^{+1} \vdash_n v^n: t$

**Main Corollary.**  $\emptyset, e \xrightarrow{0} d$  and  $\emptyset \vdash_0 e: t$  imply exist  $\mu$  and  $v^0$  and  $\Sigma$  such that  $d \equiv (\mu, v^0)$  and  $\Sigma \vdash_0 v^0: t$

## Selection of evaluation rules I

$$\mu_0, x \xrightarrow{n+1} \mu_0, x \quad \mu_0, \lambda x.e \xrightarrow{0} \mu_0, \lambda x.e \quad \frac{\mu_0, e \xrightarrow{n+1} \mu_1, v}{\mu_0, \lambda x.e \xrightarrow{n+1} \mu_1, \lambda x.v}$$

$$\frac{\mu_0, e_1 \xrightarrow{0} \mu_1, \bullet \lambda x.e \quad \mu_1, e_2 \xrightarrow{0} \mu_2, v \quad \mu_2, \bullet (e[x := v]) \xrightarrow{0} d}{\mu_0, e_1 e_2 \xrightarrow{0} d}$$

$$\frac{\mu_0, e \xrightarrow{0} \mu_1, \bullet l}{\mu_0, !e \xrightarrow{0} \mu_1, v} \mu_1(l) \equiv v \quad \frac{\mu_0, e_1 \xrightarrow{0} \mu_1, \bullet l \quad \mu_1, e_2 \xrightarrow{0} \mu_2, \bullet v}{\mu_0, e_1 := e_2 \xrightarrow{0} \mu_2 \{l = \bullet v\}, l} \quad l \in \text{dom}(\mu_1)$$

## Selection of evaluation rules II

$$\frac{\mu_0, e \xrightarrow{n+1} \mu_1, v}{\mu_0, \langle e \rangle \xrightarrow{n} \mu_1, \langle v \rangle}$$

$$\frac{\mu_0, e \xrightarrow{0} \mu_1, \bullet \langle v \rangle}{\mu_0, \tilde{e} \xrightarrow{1} \mu_1, \bullet v}$$

$$\frac{\mu_0, e \xrightarrow{n+1} \mu_1, v}{\mu_0, \tilde{e} \xrightarrow{n+2} \mu_1, \tilde{v}}$$

$$\mu_0, \%e \xrightarrow{n+1} \mu_0, \%e$$

$$\frac{\mu_0, e \xrightarrow{0} \mu_1, \bullet \langle v \rangle \quad \mu_1, \bullet v \downarrow \xrightarrow{0} d}{\mu_0, \text{run } e \xrightarrow{0} d} \text{ demotion}$$

$$\frac{\mu_0, e_1 \xrightarrow{0} \mu_1, \bullet v \quad \mu_1, e_2[x := \bullet v] \xrightarrow{0} d}{\mu_0, \text{let } x = e_1 \text{ in } e_2 \xrightarrow{0} d}$$

$$\frac{\mu_0, e \xrightarrow{0} \mu_1, \bullet v}{\mu_0, \bullet e \xrightarrow{0} \mu_1, \bullet v}$$

$$\frac{\mu_0, e \xrightarrow{0} \mu_1, \bullet v}{\mu_0, (x)e \xrightarrow{0} \mu_1, \bullet v}$$

## Relation to separate compilation and linking [Car97@POPL]

- $x_i: t_i^1 \vdash_1 e: t$  program fragment
- $b_i: \langle t_i \rangle^0 \vdash_0 be = \langle e[x_i := \sim b_i] \rangle: \langle t \rangle$  separate compilation of  $e$  in a binary  $be$
- $be[b_i := \langle v_i^1 \rangle] \xrightarrow{0} \langle v^1 \rangle$  linking/substitution of binaries
- $\emptyset \vdash_0 \text{run } be: t$  execution of complete program/fully linked binary

## Future work

- more refined operational semantics: environments, regions
- ? experimental MetaML compiler by adapting OCAML compiler
- code types for an object language  $L$ :  $\langle t \rangle_L \quad \langle e \rangle_L \quad \sim_e \quad \%_L e \quad \text{run } e$
- improved type system: qualified kinds, polymorphic types, sub-typing  
 $[t] \leq t \quad c \leq [c] \quad [t_1 \rightarrow t_2] \leq [t_1] \rightarrow [t_2] \quad [\langle t \rangle] \leq \langle [t] \rangle$
- staged type inference [SSP98@POPL] for flexibility
- ? intensional analysis of code in HOAS.

## References

- [Car97] L.Cardelli. Program fragments, linking and modularization. POPL
- [CM00] C.Calcagno, E.Moggi. Multi-stage imperative languages: a conservative extension result. SAIG
- [CMT00] C.Calcagno, E.Moggi, W.Taha. Closed types as a simple approach to safe imperative multi-stage programming. ICALP
- [Dav96] R.Davies. A temporal-logic approach to binding-time analysis. LICS
- [DP96] R.Davies, F.Pfenning. Modal analysis of staged computation. POPL
- [KCC99] S.Kamin, M.Callahan, L.Clausen. Lightweight and generative components I: source-level components. GCSE
- [KCC00] S.Kamin, M.Callahan, L.Clausen. Lightweight and generative components II: binary-level components. SAIG
- [MetaML] MetaML. <http://www.cse.ogi.edu/PacSoft/projects/metaml/>. OGI
- [SSP98] M.Shields, T.Sheard, S.Peyton-Jones. Dynamic typing as staged type inference. POPL