On Cross-Stage Persistence in Multi-Stage Programming

Yuichiro Hanada and Atsushi Igarashi

Graduate School of Informatics,
Kyoto University

June 4, 2014

What is Multi-stage Programming (MSP)?

A programming paradigm in which we can generate and run code fragments at runtime.

Applications

- DSLs [Taha ’04]
- Specilizing dynamic programming algorithms

Program Residualization

Generated code fragments can be executed

- by the very program that has generated it
- by another program

Program residualization : Serializing code fragments in order to execute it later

Styles of Multi-stage Programming

How is code represented?

- Strings
  - 'eval' function in scripting languages
- ASTs
  - Lisp with macros
- ASTs + code types
  - Scala with Lightweight Modular Staging library
  - MetaOCaml [Calcagno et al. '03]

An example program of MetaOCaml

$\# \ \textbf{let} \ a = \underline{\langle 1 + 2 \rangle}$ (* Bracket *)

$val \ a$: int code $= \langle 1 + 2 \rangle$

$\# \ \textbf{let} \ b = \langle \underline{\tilde{} a} * 2 \rangle$ (* Escape *)

$val \ b$: int code $= \langle (1 + 2) * 2 \rangle$

$\# \ \underline{\text{run} \ b}$ (* Run *)

– : int 6

Cross-Stage Persistence (CSP)

Refererence to a variable defined outside of brackets

$$\# \ \textbf{let}\ a = \langle (\underline{\%\,\mathrm{sqrt}})\ (2+2) \rangle$$

$$\# \ \mathrm{run}\ a$$

$$\longrightarrow 2$$

Residualization with CSP

```
# let h  =  open_in "a.txt"
# let c  =  ⟨(% input_line) (% h)⟩
# run c
(* returns first line of "a.txt" *)
# run c
(* returns the next line *)
# print_code c                              (* ??? *)
```

Writing the "print_code" function is not always feasible.

Our Goal

Rejecting unsafe residualization statically !

$\langle 1 + 2 \rangle$ (* Residualizable *)

let $h =$ open_in "a.txt" **in**
$\langle \%(\text{input\_line})\ \%\ h \rangle$ (* Not Residualizable *)

Why is writing "print_code" difficult?

- All values can be embedded by CSP
- Syntactic representations needed to residualize the code
  - Integers → OK
  - Functions → not always feasible
  - File handlers → impossible

We need syntactic representations of all values to mix residualization and CSP!

Our approach : $\lambda^{\triangleright\%}$

An extension of $\lambda^{\triangleright}$ [Tsukada&Igarashi '09]

- All core features of MetaOCaml
- Type safety
  - for programs written by programmers
  - for programs generated at run time
- Type-safe residualization

Ideas : Introducing residualizable code types

- Distinction between two kinds of code types
  - One is residualizable
  - Another can be used with CSP

An example program of $\lambda^{\triangleright\%}$

$\lambda^{\triangleright\%}$	MetaOCaml
$\#\ \textbf{let}\ a = \blacktriangleright_\alpha (1+2)$	$\textbf{let}\ a = \langle 1+2 \rangle$
$\longrightarrow^* \blacktriangleright_\alpha (1+2)$	
$\#\ \textbf{let}\ b = \blacktriangleright_\alpha (\blacktriangleleft_\alpha a * 2)$	$\textbf{let}\ b = \langle \tilde{}a * 2 \rangle$
$\longrightarrow^* \blacktriangleright_\alpha ((1+2) * 2)$	

An example program of $\lambda^{\triangleright\%}$

$\lambda^{\triangleright\%}$	MetaOCaml
$\# \ \mathbf{let}\ a = \ \Lambda\beta.(\blacktriangleright_\beta(1+2))$	$\mathbf{let}\ a = \langle 1+2 \rangle$
$\quad \longrightarrow^* \Lambda\beta.(\blacktriangleright_\beta(1+2))$	
$\# \ \mathbf{let}\ b = \ \Lambda\alpha.\blacktriangleright_\alpha(\blacktriangleleft_\alpha(a\ \alpha) * 2)$	$\mathbf{let}\ b = \ \langle \tilde{}a * 2 \rangle$
$\quad \longrightarrow^* \Lambda\alpha.\blacktriangleright_\alpha(\blacktriangleleft_\alpha \blacktriangleright_\alpha(1+2) * 2)$	
$\quad \longrightarrow^* \Lambda\alpha.\blacktriangleright_\alpha((1+2) * 2)$	
$\# \ b\ \varepsilon$ (* ε : empty sequence *)	$.!\ b$
$\quad \longrightarrow^* (\blacktriangleright_\alpha((1+2) * 2))[\alpha := \varepsilon]$	
$\quad \longrightarrow^* (1+2) * 2 \longrightarrow^* 6$	

An example program of $\lambda^{\triangleright\%}$ with CSP

$$\textbf{let } f = \lambda x : \textbf{int}.x * 2 \textbf{ in}$$

$$\blacktriangleright_\alpha(\%_\alpha(f\ 1) + (\%_\alpha\ f)\ 1)$$

$$\longrightarrow_s^* \blacktriangleright_\alpha(\%_\alpha(1 * 2) + (\%_\alpha(\lambda x : \textbf{int}.x * 2))\ 1)$$

$$\longrightarrow_s^* \blacktriangleright_\alpha(\%_\alpha\ 2 + \%_\alpha(\lambda x : \textbf{int}.x * 2)\ 1)$$

The former application is evaluated but the latter is not.

An example program of $\lambda^{\triangleright\%}$ with CSP

$$\textbf{let } f = \lambda x : \textbf{int}.x * 2 \textbf{ in}$$

$$(\Lambda\alpha.\blacktriangleright_\alpha((\%_\alpha\, f)\ 1))\ \varepsilon$$

$$\longrightarrow^* (\Lambda\alpha.\blacktriangleright_\alpha((\%_\alpha(\lambda x : \textbf{int}.x * 2))\ 1))\ \varepsilon$$

$$\longrightarrow^* (\lambda x : \textbf{int}.x * 2)\ 1$$

$$\longrightarrow^* 2$$

The substitution removes the "$\%_\alpha$".

Syntax of $\lambda^{\triangleright\%}$

Transition Variable (TV) ::= $\alpha, \beta, \gamma \cdots$

- Level of nested brackets
- c.f. Environment classifiers [Taha&Nielsen '03]

Terms

$M ::= x \mid \lambda x : \tau.M \mid M_1\ M_2 \mid \blacktriangleright_\alpha M \mid \blacktriangleleft_\alpha M$
$\mid \Lambda\alpha.M \mid M\ A$ (A = a sequence of TV) $\mid \%_\alpha M$

Syntax of $\lambda^{\triangleright\%}$ (2)

Terms

$$M ::= x \mid \lambda x : \tau.M \mid M_1\ M_2 \mid \blacktriangleright_\alpha M \mid \blacktriangleleft_\alpha M$$
$$\mid \Lambda\alpha.M \mid M\ A \text{ (A = a sequence of TV)} \mid \%_\alpha M$$

- $\Lambda\alpha.M$ = a binder of TV
- $M\ A$ = an application to a sequence of TVs
- $\varepsilon \in \mathrm{TV}^*$ = a special symbol for empty sequence

$$(\Lambda\alpha.\blacktriangleright_\alpha M)\ (\beta\gamma) \longrightarrow \blacktriangleright_{\beta\gamma} M$$
$$(\Lambda\alpha.\blacktriangleright_\alpha M)\ (\varepsilon) \longrightarrow M$$

Full reduction and staged reduction

- Full reduction
  - Substitution-based
  - Only 3 rules
    - $(\lambda x : \tau.M)\ N \longrightarrow M[x := N]$
    - $(\Lambda\alpha.M)\ A \longrightarrow M[\alpha := A]$
    - $\blacktriangleleft_\alpha \blacktriangleright_\alpha M \longrightarrow M$
- Staged reduction (call-by-value, deterministic)
  - Evaluation contexts

Type system of $\lambda^{\triangleright\%}$

Types

$\tau ::= b \mid \tau \to \tau \mid \triangleright_\alpha \tau \mid \forall \alpha.\tau \mid \forall^\varepsilon \alpha.\tau$

- $\triangleright_\alpha \tau$: a type of code of type τ
- $\forall \alpha.\tau, \forall^\varepsilon \alpha.\tau$: types for $\Lambda\alpha.M$
    - $\forall\alpha$: $\blacktriangleright_\alpha M$ is residualizable
    - $\forall^\varepsilon\alpha$: $\blacktriangleright_\alpha M$ is not residualizable
        - $\blacktriangleright_\alpha(\cdots \%_\alpha M \cdots)$ is OK

Type system of $\lambda^{\triangleright\%}$ (2)

Type judgment

$\Gamma; \Delta \vdash^{A} M : \tau$

- Δ : a set of classifiers introduced by $\forall^{\varepsilon}\alpha$
- A : current stage (a sequence of classifiers)

$$\vdash^{A} \Lambda\alpha.\blacktriangleright_{\alpha}(1+2) : \forall\alpha.\triangleright_{\alpha}\ \textbf{int}$$
$$\vdash^{A} (\Lambda\alpha.\blacktriangleright_{\alpha}(1+2))\ \varepsilon : \textbf{int}$$
$$\nvdash^{A} \blacktriangleright_{\alpha}((\%_{\alpha}\ 1)+2) : \triangleright_{\alpha}\textbf{int}$$
$$\emptyset; \{\alpha\} \vdash^{A} \blacktriangleright_{\alpha}((\%_{\alpha}\ 1)+2) : \triangleright_{\alpha}\textbf{int}$$

Typing rules of $\lambda^{\triangleright\%}$

$$\frac{\Gamma; \Delta \vdash^{A\alpha} M : \tau}{\Gamma; \Delta \vdash^{A} \blacktriangleright_\alpha M : \triangleright_\alpha \tau} \ (\blacktriangleright)$$

$$\frac{\Gamma; \Delta \vdash^{A} M : \triangleright_\alpha \tau}{\Gamma; \Delta \vdash^{A\alpha} \blacktriangleleft_\alpha M : \tau} \ (\blacktriangleleft)$$

$$\frac{\Gamma; \Delta \vdash^{A} M : \tau \qquad \alpha \in \Delta}{\Gamma; \Delta \vdash^{A\alpha} \%_\alpha M : \tau} \ (\%)$$

Typing rules of $\lambda^{\triangleright\%}$ (2)

$$\frac{\Gamma; \Delta \vdash^{A} M : \tau \qquad \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(A) \cup \Delta}{\Gamma; \Delta \vdash^{A} \Lambda\alpha.M : \forall\alpha.\tau} \ (\textsc{Gen})$$

$$\frac{\Gamma; \Delta \vdash^{A} M : \forall\alpha.\tau}{\Gamma; \Delta \vdash^{A} M\, B : \tau[\alpha := B]} \ (\textsc{Ins})$$

$$\frac{\begin{array}{c}\Gamma;\ \Delta \cup \{\alpha\} \vdash^{A} M : \tau \\ \alpha \notin \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(A) \cup \Delta\end{array}}{\Gamma; \Delta \vdash^{A} \Lambda\alpha.M : \forall^{\varepsilon}\alpha.\tau} \ (\textsc{GenE})$$

$$\frac{\Gamma;\ \Delta \vdash^{A} M : \forall^{\varepsilon}\alpha.\tau \qquad \beta \in \Delta \text{ whenever } \beta \in B}{\Gamma;\ \Delta \vdash^{A} M\, B : \tau[\alpha := B]} \ (\textsc{InsE})$$

Type-safe residualization

Theorem : Type-safe residualization

If $\vdash^{\varepsilon} M : \triangleright_{\alpha}\tau$ is derivable then

- $M \longrightarrow_{s}^{*} \blacktriangleright_{\alpha} v^{\alpha}$ for some v^{α}
  - v^{α} : a value at stage α
- $\vdash^{\varepsilon} v^{\alpha} : \tau$ is derivable.

Well-typed programs of code types yield residualizable programs

Other properties of $\lambda^{\triangleright\%}$

- Staged reduction $\subset$ Full reduction
- Subject Reduction
- Strong Normalization
- Confluence
- Progress
  - under staged reduction

Related Work

- λ^{BN} [Benaissa et al. '99]
  - Explict CSP operator "up"
    - for any kinds of values
    - as if any value has its syntactic representation
- λ^{α} [Taha&Nielsen '03]
  - Environment classifiers
  - No distiction between residualizable and nonresidualizable code

Conclusion

New typed multi-stage calculus $\lambda^{\triangleright\%}$

- All core features of MetaOCaml
- Full reduction and staged reduction
- Distinction between residualizable code types and non-residualizable code types
- Type-safe residualization
- (Subtyping for two kinds of code types)

CSP vs. Lifting

- Cross-Stage Persistence
  - Just a syntactic marker waiting for run to dissolve the surrounding brackets
- Lifting (in Partial Evaluation)
  - Convering a value into its syntactic representation