Multi-Stage Programming

Introduction

Multi-Stage Programming is a paradigm for developing generic software, designed to address a number of problems with dynamic code generation.

Program generation - motivation

- Code reuse
- Developer productivity
- Code reliability and maintainability
- Performance

Program generation - problems

The key problem is representing the dynamically generated code. There are two popular approaches:

- Strings
- Data types (abstract syntax trees)

Unfortunately, both approaches have disadvantages.

Dynamic code representation - strings

- No automatic way of guaranteeing syntactic correctness.
- No guarantee of being well-typed.
- Advantage: concise and usually clearly understandable.

Dynamic code representation – syntax trees

- Advantage: Guarantee syntactic correctness.
- No guarantee of being well-typed.
- Verbose notation, can be hard to both write and read.

MetaOCaml

MetaOCaml is an extension of Ocaml that provides constructs for writing multi-stage programs.

Available at:
http://www.cs.rice.edu/~taha/MetaOCaml/

Basic concepts of MSP in MetaOCaml

- Brackets
- Escape
- Run

Basic concepts of MSP - Brackets

Brackets can be inserted around any expression, changing its type and delaying its evaluation.

```
# let a = 1+2;;
val a : int = 3
# let a = .<1+2>.;;
val a : int code = .<1+2>.
```

Basic concepts of MSP - Brackets

Variables will be evaluated and defined functions will be stored as cross-stage persistent values.

```
# let a = 5;;
val a : int = 5
# let f x = x + 2;;
val f : int -> int = <fun>
# let b = .< f a >. ;;
val b : int code =.<(((* cross-
  stage persistent value (as id: f)
  *)) 5)>.
```

Basic concepts of MSP – Escape

Used for combining smaller fragments of code into larger ones.

```
# let a = .<1+2>.;;
val a : int code = .<1+2>.
# let b = .<.~a * .~a>. ;;
val b : int code = .<(1 + 2) * (1 +
  2)>.
```

Basic concepts of MSP - Run

Used to compile and execute the dynamically generated code.

```
# let a = .<1+2>.;;
val a : int code = .<1+2>.
# let c = .! a;;
val c : int = 3
```

Cross-stage persistent values - example

```
# let a = 5;;
val a : int = 5
# let f = fun x -> x+2;;
val f : int -> int = <fun>
# let code = .< ((f a)+7)>.;;
val code : ('a, int) code = .<((((* cross-
  stage persistent value (as id: f) *)) 5)
  + 7)>.
# let f = fun x -> x-10;;
val f : int -> int = <fun>
# .! code;;
- : int = 14
```

Cross-stage persistent values - details

- Standard library functions are stored by name.

```
# let code = .< exp (3.5) >.;;
val code : float code = .<(exp
  3.5)>.
```

- Other functions are stored as cross-stage persistent values.

```
let code = .<List.length 1;2;3]>.;;
val code : int code = .<(((* cross-
  stage persistent value (as id:
  List.length) *)) [1; 2; 3])>.
```

Example: expotentiation

```
let rec power (n, x) =
  match n with
     0 -> 1
   | n -> x * (power (n-1, x));;

let power2 = fun x -> power (2,x);;
```

Example: expotentiation

```
let rec power (n, x) =
  match n with
     0 -> 1
   | n -> x * (power (n-1, x));;

let power2 = fun x -> power (2,x);;
```

Issue: calling `power2` will cause two recursive calls of `power` each time.

Example: expotentiation

```
let rec power (n, x) =
  match n with
     0 -> .<1>.
   | n -> .< .~x * .~(power (n-1, x))>.;;

let power2 = fun x -> power (2,x);;
```

Applying staging annotations to `power`. The type of `power` is now `int -> code -> int code`

Example: expotentiation

```
let rec power (n, x) =
  match n with
     0 -> .<1>.
   | n -> .< .~x * .~(power (n-1, x))>.;;

let power2 = .! .<fun x -> .~(power
  (2,.<x>.))>.;;
```

Applying staging annotations to `power2`. The type of `power2` is unchanged.

Example: expotentiation

```
let power2 = .! .<fun x -> .~(power
   (2,.<x>.))>.;;
```

It’s worth noting that when this definition of `power2` is evaluated, it will be compiled into a static piece of code that behaves exactly the same as defining:

```
let power2 = fun x -> x*x*1;;
```

Avoiding accidental name capture

MetaOCaml automatically renames bound variables that occur inside the code. The purpose is to avoid causing behavior that differs from unstaged code.

Avoiding accidental name capture - example

```
# let rec h n z =
  if n=0 then z
         else (fun x -> (h (n-1) x+z)) n;;
val h : int -> int -> int = <fun>

# h 3 1;;
- : int = 7
```

Avoiding accidental name capture - example

```
# let rec h n z =
  if n=0 then z
         else .<(fun x -> .~(h (n-1) .<x+
         .~z>.)) n>.;;
val h : int -> int code -> int code = <fun>

# h 3 .<1>.;;
- : int code = .<(fun x_1 -> (fun x_2 ->
  (fun x_3 -> x_3 + (x_2 + (x_1 + 1))) 1) 2)
  3>.
```

Avoiding accidental name capture - example

Without automatic variable renaming:

```
# h 3 .<1>.;;
- : int code = .<(fun x -> (fun x -> (fun x
  -> x + (x + (x + 1))) 1) 2) 3>.
```

Which evaluates to 4 rather than 7.

How to write MSP programs?

- Write a single-stage program.
- Study and analyze the program.
- Find fragments of the code that can be staged.
- Add staging annotations to specify the evaluation order.

Example of writing an MSP program

The example program is an interpreter for a toy programming language called LINT (Little Integer).

This language supports integer arithmetic, conditionals, and recursive functions.

LINT programs consist of a series of definitions of single-variable functions followed by a single expression to be evaluated.

Example LINT program

```
fact (x) = if (x = 0) then 1 else x*(fact (x-1))
fact (10)
```

Why an interpreter?

A typical problem with writing language interpreters is the performance overhead required to execute programs. However, a staged interpreter will translate the LINT program into a MetaOCaml program that can then be executed without additional overhead.

Defining LINT in MetaOCaml

```
type exp = Int of int
         | Var of string
         | App of string * exp
         | Add of exp * exp
         | Sub of exp * exp
         | Mul of exp * exp
         | Div of exp * exp
         | Ifz of exp * exp * exp
type def = Declaration of string *
  string * exp
type prog = Program of def list * exp
```

Syntax tree of the example LINT program

```
Program ([Declaration
  ("fact","x", Ifz(Var "x",
     Int 1,
     Mul(Var"x„,(App ("fact",
         Sub(Var x",Int 1)))))))
],
App ("fact", Int 10))
```

Defining LINT in MetaOCaml - environtment

```
exception Yikes
let env0 = fun x -> raise Yikes
let fenv0 = env0
let ext env x v = fun y -> if
  x=y then v else env y
```

Unstaged interpreter

```
let rec eval e env fenv =
match e with
  Int i -> i
| Var s -> env s
| App (s,e2) -> (fenv s)(eval e2 env fenv)
| Add (e1,e2) -> (eval e1 env fenv)+(eval e2 env fenv)
| Sub (e1,e2) -> (eval e1 env fenv)-(eval e2 env fenv)
| Mul (e1,e2) -> (eval e1 env fenv)*(eval e2 env fenv)
| Div (e1,e2) -> (eval e1 env fenv)/(eval e2 env fenv)
| Ifz (e1,e2,e3) -> if (eval e1 env fenv)=0
                    then (eval e2 env fenv)
                    else (eval e3 env fenv)
```

Unstaged interpreter

```
let rec peval p env fenv=
    match p with
     Program ([],e) -> eval e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        let rec f x = eval e1 (ext env s2 x) (ext
  fenv s1 f)
        in peval (Program(tl,e)) env (ext fenv s1 f)
```

Staged interpreter

```
let rec eval2 e env fenv =
match e with
  Int i -> .<i>.
| Var s -> env s
| App (s,e2) -> .<.~(fenv s).~(eval2 e2 env fenv)>.
| Add (e1,e2) -> .<.~(eval2 e1 env fenv)+ .~(eval2 e2 env
   fenv)>.
| Sub (e1,e2) -> .<.~(eval2 e1 env fenv)- .~(eval2 e2 env
   fenv)>.
| Mul (e1,e2) -> .<.~(eval2 e1 env fenv)* .~(eval2 e2 env
   fenv)>.
| Div (e1,e2) -> .<.~(eval2 e1 env fenv)/ .~(eval2 e2 env
   fenv)>.
| Ifz (e1,e2,e3) -> .<if .~(eval2 e1 env fenv)=0
                      then .~(eval2 e2 env fenv)
                      else .~(eval2 e3 env fenv)>.
```

Staged interpreter

```
let rec peval2 p env fenv=
    match p with
     Program ([],e) -> eval2 e env fenv
    |Program (Declaration (s1,s2,e1)::tl,e) ->
        .<let rec f x = .~(eval2 e1 (ext env s2
  .<x>.) (ext fenv s1 .<f>.))
          in .~(peval2 (Program(tl,e)) env (ext fenv
  s1 .<f>.))>.
```

Result of interpretting the example program

```
.<let rec f = fun x -> if x = 0
  then 1 else x * (f (x - 1)) in
  (f 10)>.
```

Bibliography

1. A Gentle Introduction to Multi-stage Programming. Walid Taha. Domain-Specific Program Generation 2003: 30-50.
2. A Gentle Introduction to Multi-stage Programming, Part II. Walid Taha. GTTSE 2007: 260-290.
3. http://www.cs.rice.edu/~taha/MetaOCaml/

Resources

- http://www.cs.rice.edu/~taha/MetaOCaml/ - MetaOCaml homepage, download is unavailable but the site has examples and useful links
- http://web.archive.org/web/20120209151915/http://www.metaocaml.org/dist/old/MetaOCaml_309_alpha_030.tar.gz - archived download of MetaOCaml
- http://okmij.org/ftp/ML/MetaOCaml.html - reimplementation of MetaOCaml, code should be cross-compatible