B. Comp Dissertation

Automated Meta-Programming to Support High-Performance OCaml Codes

Arnold Christopher Koroa

Department of Computer Science, School of Computing

National University of Singapore

2014/2015

B. Comp Dissertation

Not Manual

???

Automated Meta-Programming to Support High-Performance OCaml Codes

/səˈpɔːt/ verb
bear all or part of the weight of

Fast!!

A Programming Language

Arnold Christopher Koroa

My Name

Department of Computer Science, School of Computing

National University of Singapore

2014/2015

My School

Meta-Programming

- Programs that manipulate other programs
  - “Code fragment objects” as data
  - Create new code
  - Modify existing code

Meta-Programming, how is it useful?

- “Free” compiler from an interpreter
  - Partial evaluation of an Domain Specific Language (DSL) interpreter to a DSL program
  - Produces code in the interpreter’s implementation language
  - Can then reuse the interpreter’s implementation language’s compiler

Meta-Programming, how is it useful?

- “Free” compiler from an interpreter

```
(* if n = 0 then 1 else x * 200 *)
Ifz (Var “n", Int 1, Mul (Var "x", Int 200))

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

Meta-Programming, how is it useful?

- “Free” compiler from an 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)>.

 => .<if n = 0 then 1 else x * 200>.
```

Meta-Programming, how is it useful?

- “Free” compiler from an interpreter
  - Results from Taha (2004)

Program	Description of Interpreter	Fact10	Fib20
(none)	OCaml implementations	100%	100%
`eval1`	Simple	1,570%	1,736%
`eval2`	Simple staged	100%	100%
`eval3`	Error handling (EH)	1,903%	2,138%
`eval4`	EH staged	417%	482%
`eval5`	CPS, EH	2,470%	2,814%
`eval6`	CPS, EH, staged	100%	100%
`eval7`	Inlining, staged	87%	85%
`eval8`	Inlining, no duplication, staged	97%	97%
`eval9`	Inlining, CPS, EH, staged	90%	85%

Meta-Programming, how is it useful?

- “Free” compiler from an interpreter
  - Results from Hermann and Langhammer (2006)

	native base program		bytecode base program	
configuration	t in sec.	speedup	t in sec.	speedup
none	259.96	1.00×	1 032.33	1.00×
simplify	135.32	1.92×	640.04	1.61×
staged*	3.98	65.24×	29.90	34.52×
simplify + staged*	2.75	94.68×	15.83	65.21×

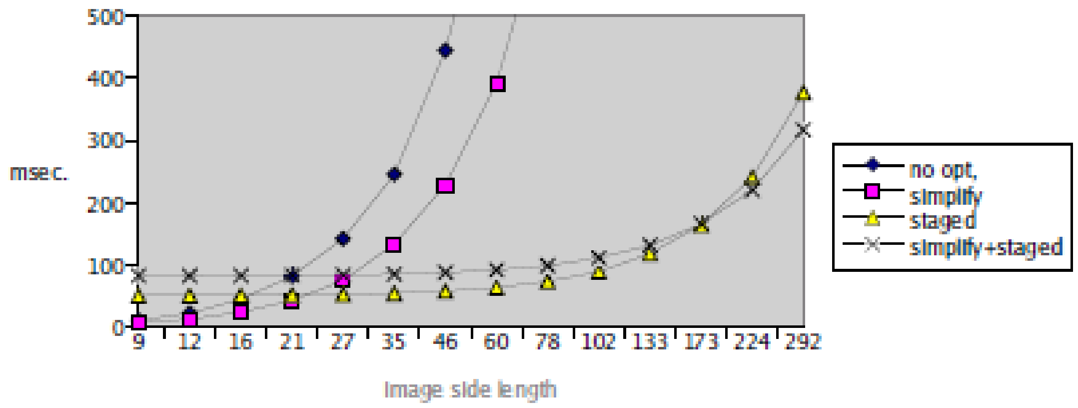
* Staged run comprises code generation and .! application.

Table 4.2: Overall execution times for input image of size 1000 × 1000.

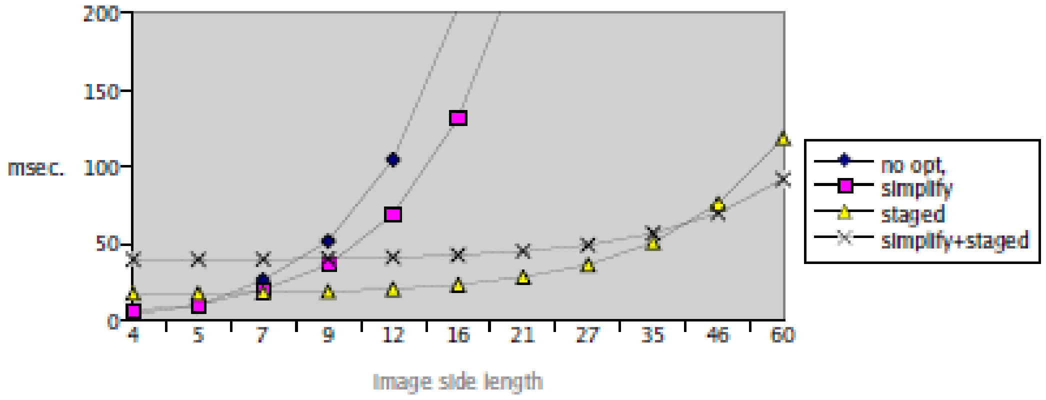
Meta-Programming, how is it useful?

- “Free” compiler from an interpreter
  - Results from Hermann and Langhammer (2006)

Native base program:



Bytecode base program:

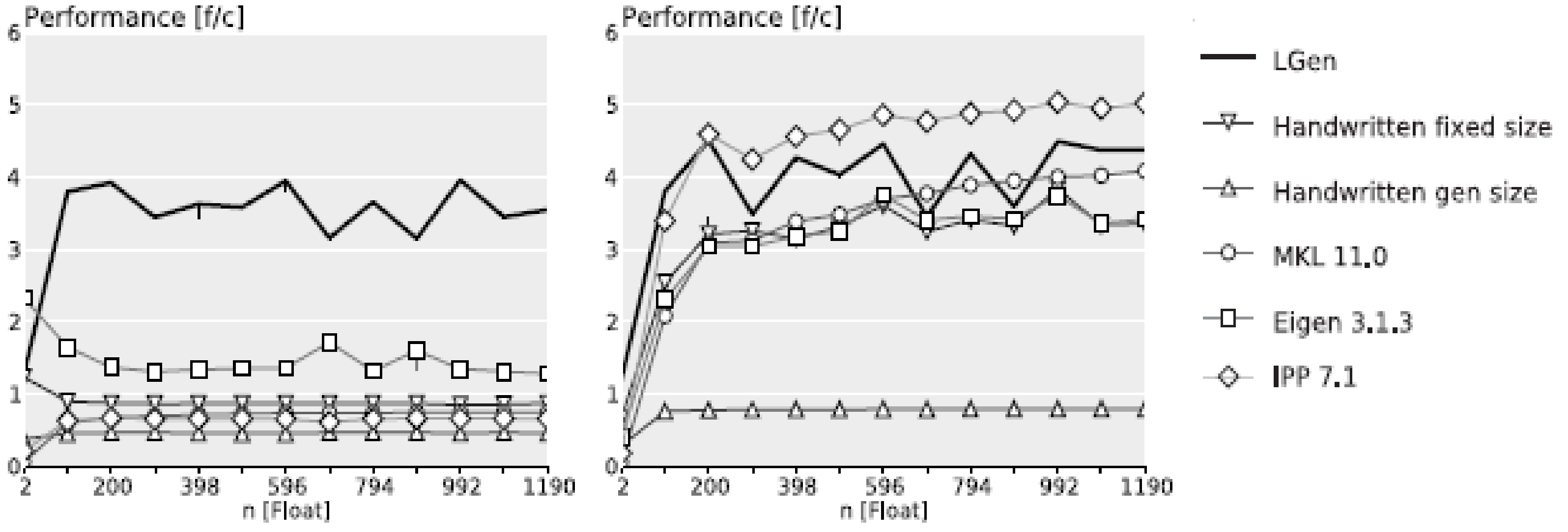


Meta-Programming, how is it useful?

- Efficient genericity and high level code
  - Adaptable code that optimizes to the problem being solved
    - input size
    - input/output type
  - Automatic optimization to platform
    - Detection and utilization of underlying hardware
      - Unrolling according to cache size
      - SSE
      - Distributed/Parallel Architecture

Meta-Programming, how is it useful?

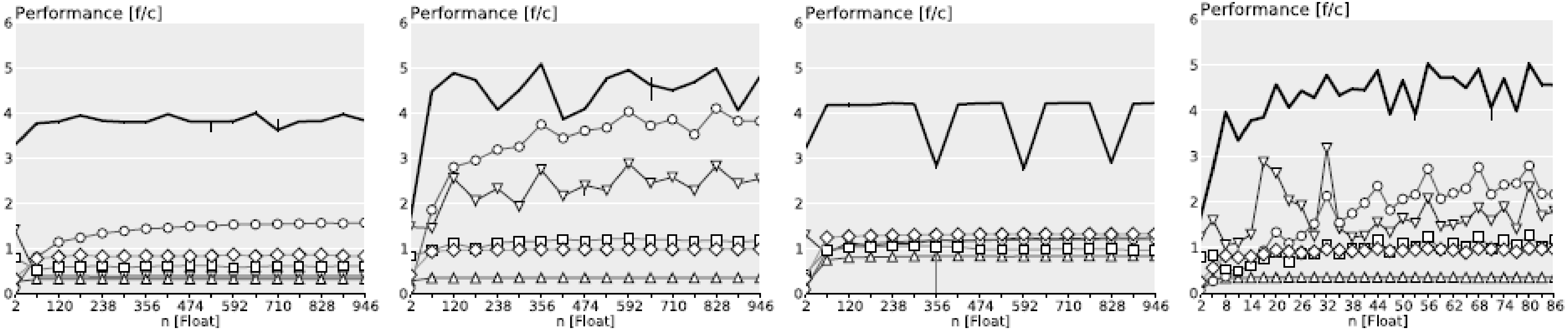
- Efficient genericity and high level code
  - Results from Spampinato and Puschel (2014)



(a) A is $n \times 4$. (b) A is $4 \times n$.

Meta-Programming, how is it useful?

- Efficient genericity and high level code
  - Results from Spampinato and Puschel (2014)



(c) A is $n \times 4$, B is 4×4. (d) A is 4×4, B is $4 \times n$. (e) A is $4 \times n$, B is $n \times 4$. (f) A is $n \times 4$, B is $4 \times n$.

Meta-Programming, how to do it?

- MetaOCaml
  - Multi-staged programming language
    - Allows the creation of delayed computations in the form of code
    - The delayed computations can produce further delayed computations
  - Based on OCaml

Meta-Programming, how to do it?

- MetaOCaml
  - Bracket `.< … >.`
    - Delay the computation inside it

```
let plus2 x = .<x + 2>.;;
# plus2 3;;
- : int code = .<3 + 2>.
```

  - Escape `.~`
    - Runs code pointed by it to produce code to be spliced

```
# .<.~(plus2 3) * .~(plus2 4)>.;;
- : int code = .<(3 + 2) * (4 + 2)>.
```

  - Run `!.`
    - Executes a delayed computation

```
# !. .<.~(plus2 3) * .~(plus2 4)>.;;
- : int = 30
```

Meta-Programming, how to do it?

- MetaOCaml

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

```
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)>.
```

?

Project Objective

To automate the staging process, enabling users to reap the benefits of multi-staged meta-programming without having to deal with the complexities of manually staging source programs

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Automated Staging of OCaml Codes

- Annotate OCaml Code
  - What function to stage
  - What static information is available
- Preprocess the annotated OCaml code
  - Extract information in annotations
  - Analyze and transform
  - Produce staged MetaOCaml code

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Translation Rules

- Simple functions with no control flows or recursions
  - `let plus x y = x + y`
  - If y is always 2

    `=> let plus x = x + 2`
  - How to stage?

    `=> let plus_staged y = .< let plus x = x + y in plus >.`
  - Sample Run: `# plus_staged 2;;`

    `- : (int -> int) code = .< let plus x = x + 2 in plus >.`

Translation Rules

```
let plus x y = x + y => let plus_staged y = .< let plus x = x + y in plus >.
```

```
<FunBody> => <FunBodyS>
-----------------------------------------------------------
let <FunName> <Args> = <FunBody>
```

$=>_{toMeta}$

```
let <FunName>_staged <StaticArgs> =
  .< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.
```

Translation Rules

```
<FunBody> => .< <FunBody> >.
```

Translation Rules

- Function with control flow
  - `let f x y = if x > 0 then x + y else x - y`
  - If y is static

    `=> let f_staged y = .< let f x = if x > 0 then x + y else x – y in f >.`
  - If x is static?

    `=> let f_staged x = .< let f y = if x > 0 then x + y else x – y in f >.`

```
# f_staged 1;;

- : (int -> int) code = .< let f y = if 1 > 0 then 0 + y else 0 – y in f >.
```

Translation Rules

- Function with control flow
  - `let f x y = if x > 0 then x + y else x - y`
  - If x is static

```
=> let f_staged x =
   let aux x y = if x > 0 then .< 0 + .~y >. else .< 0 – .~y >.
   in .< let f y = .~(aux x .<y>.) in f >.


# f_staged 1;;
- : (int -> int) code = .< let f y = 1 + y in f >.
```

Translation Rules

- Function with control flow
  - Nested control structures?
  - `let f x y = if x = 0 then (if y = 0 then true else false) else false`
  - If y is static

```
=> let f_staged y =
      let aux x y = if y = 0 then .<true>. else .<false>. in
     .< let f = if x = 0 then .~(aux x y) else false in f
```

Translation Rules

- Function with control flow
  - Nested control structures?
  - `let f x y = if x = 0 then (if y = 0 then true else false) else false`
  - If both x and y are static

```
=> let f_staged x y =
     let aux_2 x y = if y = 0 then .< true >. else .< false >. In
    let aux_1 x y = if x = 0 then .~(aux_2 x y) else .< false >. In
    .< let f = .~(aux_1 x y) in f >.
```

Translation Rules

<FunBody> => ([], .< <FunBody> >.)

——

<FunBody> => (**[<Aux$_1$>, …, <Aux$_n$>]**, <FunBodyS>)

---

let <FunName> <Args> = <FunBody>

=>$_{\text{toMeta}}$

let <FunName>_staged <StaticArgs> =

let <AuxName$_1$> = <AuxBody$_1$>

and …

and <AuxName$_n$> = <AuxBody$_n$> in

.< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.

Translation Rules

For if-then-else structure with statically computable condition

```
                        isStatic(CondExp)

<ThenBody> => (ThenAuxList, <ThenBodyS>)        <ElseBody> => (ElseAuxList, <ElseBodyS>)
______________________________________________________________________________________
             if <CondExp> then <ThenBody> else <ElseBody>

                                 =>

( ThenAuxList @ ElseAuxList @
  [if <CondExp>                        ,  .< .~(aux <Args[DynArg->.<DynArg>.]>) >.
     then .< .~(<ThenBodyS[DynArg->.~DynArg]>) >.
     else .< .~(<ElseBodyS[DynArg->.~DynArg]>) >.]                                  )
```

Translation Rules

For if-then-else structure with non-statically computable condition

```
                 !isStatic(CondExp)       <CondBody> => (CondAuxList, <CondBodyS>)

    <ThenBody> => (ThenAuxList, <ThenBodyS>)        <ElseBody> => (ElseAuxList, <ElseBodyS>)
_____________________________________________________________________________________________

                        if <CondExp> then <ThenBody> else <ElseBody>

                                          =>

(                                                          if .~(<CondExpS>)

     CondAuxList @ ThenAuxList @ ElseAuxList     ,           then .< .~(<ThenBodyS>) >.

                                                             else .< .~(<ElseBodyS>) >.    )
```

Translation Rules

- Function with recursion
  - `let rec pow x n = if n = 0 then 1 else x * pow x (n - 1)`
  - Change recursive call to the optimized function call
    - Omitted here for brevity

Translation Rules

- Function that uses other staged function
  - `let rec ff y m =`

    ```
    if y = 0 then 0 else (pow y m) + (ff (y - 1) m)
    ```

  - If m is static and pow is staged over its second argument

```
=> let ff_staged m =
     let pow = !. .< .~(pow_staged m) >. in
     .< let rec ff y =
          if y = 0 then 0 else (pow y) + (ff (y - 1)) in ff >.
```

Translation Rules

- Function that uses other staged function

```
=> let ff_staged m =
     let pow = !. .< .~(pow_staged m) >. in
     .< let rec ff y =
          if y = 0 then 0 else (pow y) + (ff (y - 1)) in ff >.

# ffstaged 3;;
- : (int -> int) code =
      .< let rec ff y =
           if y = 0 then 0 else ((* CSP pow *) y) + (ff (y - 1))
         in ff >.
```

Translation Rules

- Function that uses other staged function

```
=> let ff_staged m =
     let pow = !. .< .~(pow_staged m) >. in
     .< let rec ff y =
          if y = 0 then 0 else (pow y) + (ff (y - 1)) in ff >.
```

- CSP = Cross-Stage Persistence
  - Values/functions from the code generator environment used in the produced code
  - Code generator environment needs to be available to the produced code
  - Assume `ff_staged` always called before the produced specialized code is used

Translation Rules

```
<FunBody> => ([<Aux_1>, …, <Aux_n>], <FunBodyS>)
────────────────────────────────────────────────
let <FunName> <Args> = <FunBody>

=>_toMeta

let <FunName>_staged <StaticArgs> =
  let <StagedFun> = !. .< .~(<StagedFun>_staged <StaticArgs>) >. in
  let <AuxName_1> = <AuxBody_1>
  and …
  and <AuxName_n> = <AuxBody_n> in
  .< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.
```

Translation Rules

- Now we know how to systematically translate OCaml code into a staged MetaOCaml code
  - Given the static information available to them
  - Staged code produces optimized function

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Source Code Annotation

- How to indicate in the source program what is static?
- OCaml Attributes
  - "decorations" attachable to OCaml AST
  - `[@id payload]` or `[@@id payload]` or `[@@@id payload]`
    - `(1 + 2) [@doSomething]`
    - `let x = 2 [@@doSomething [data1;data2]]`

- Available since OCaml 4.02.1

Source Code Annotation

- How to indicate in the source program what is static?
  - `[@@static [statVar1; …; statVarn]`

```
let rec pow x n =
  if n = 0 then 1 else x * (pow x (n - 1))
[@@static [n]]
```

Source Code Annotation

- How to indicate in the source program what is static?
  - `[@@static [statVar1; …; statVarn]]`

```
let rec pow x n =
  if n = 0 then 1 else x * (pow x (n - 1))
```

`[@@static []] [@@static [x]] [@@static [n]] [@@static [x;n]]`

Source Code Annotation

- How to indicate in the source program if we use other (automatically) staged function?
  - `[@static.use]`

```
let rec ff x n =
  if x = 0 then 0 else pow x n [@static.use] + ff (x - 1) n
[@@static [n]]
```

- Necessary because sometimes we don't want to use the static version

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Source Code Preprocessing

- How to process the annotated code?
- ppx Preprocessor
  - Mapper from OCaml AST to OCaml AST
  - Default mapper
    - Do deep identity mapping
    - Can be overridden at points where it takes in different OCaml AST nodes

Source Code Preprocessing

- Hooking into the default mapper
  - `let plus x y = x + y`

```
structure
 [ structure_item (foo.ml[1,0+0]..[1,0+20])
     Pstr_value Nonrec
     [ <def>
         pattern (foo.ml[1,0+4]..[1,0+8])
           Ppat_var "plus" (foo.ml[1,0+4]..[1,0+8])

         expression (foo.ml[1,0+9]..[1,0+20]) ghost
           Pexp_fun ""
           None
           pattern (foo.ml[1,0+9]..[1,0+10])
             Ppat_var "x" (foo.ml[1,0+9]..[1,0+10])
           expression (foo.ml[1,0+11]..[1,0+20]) ghost
             Pexp_fun ""
             None
             pattern (foo.ml[1,0+11]..[1,0+12])
               Ppat_var "y" (foo.ml[1,0+11]..[1,0+12])
             expression (foo.ml[1,0+15]..[1,0+20])
               Pexp_apply
               expression (foo.ml[1,0+17]..[1,0+18])
                 Pexp_ident "+" (foo.ml[1,0+17]..[1,0+18])
               [ <label> ""
                   expression (foo.ml[1,0+15]..[1,0+16])
                     Pexp_ident "x" (foo.ml[1,0+15]..[1,0+16])
                 <label> ""
                   expression (foo.ml[1,0+19]..[1,0+20])
                     Pexp_ident "y" (foo.ml[1,0+19]..[1,0+20])
               ]
     ]
 ]
```

Source Code Preprocessing

- Hooking into the default mapper

```
let toMeta_mapper argv =
  { default_mapper with
    structure = fun mapper structure_item_list -> … }
```

Source Code Preprocessing

- Hooking into the default mapper

```
match structure_item with
  {pstr_desc = Pstr_value (_, _)} ->
      if hasToMetaAnnot structure_item
        then (* do generation of staged code *)
        else [default_mapper.structure_item mapper structure_item]
  | _ -> [default_mapper.structure_item mapper structure_item]
```

Source Code Preprocessing

- Extracting information and annotations
  - `let f x = x + 1 [@@static [x]]`

```
[{pstr_desc =
   Pstr_value (Nonrecursive,
    [{pvb_pat = {ppat_desc = Ppat_var {txt = "f"}};
       pvb_expr =
        {pexp_desc =
          Pexp_fun ("", None, {ppat_desc = Ppat_var {txt = "x"}},
           {pexp_desc =
             Pexp_apply ({pexp_desc = Pexp_ident {txt = Lident "+"}},
              [("", {pexp_desc = Pexp_ident {txt = Lident "x"}});
               ("", {pexp_desc = Pexp_constant (Const_int 1)})])})};
     pvb_attributes =
      [({txt = "static"},
        PStr [{pstr_desc =
         Pstr_eval
          ({pexp_desc =
             Pexp_construct ({txt = Lident "::"},
              Some {pexp_desc =
               Pexp_tuple
                [{pexp_desc = Pexp_ident {txt = Lident "x"}};
                 {pexp_desc = Pexp_construct ({txt = Lident "[]"}, None)}]})},
...)}])]}])}]
```

Source Code Preprocessing

- Extracting information and annotations
  - OCaml Structure item (let-bound function defs): `Pstr_value`
  - Control structures: `Pexp_ifthenelse, Pexp_match`
  - Function application: `Pexp_apply`
  - Annotations: `pvb_attributes, pexp_attributes`

Source Code Preprocessing

- Building the staged function

```
                              isStatic(CondExp)
   <ThenBody> => (ThenAuxList, <ThenBodyS>)        <ElseBody> => (ElseAuxList, <ElseBodyS>)
--------------------------------------------------------------------------------------------
                     if <CondExp> then <ThenBody> else <ElseBody>
                                        =>
( ThenAuxList @ ElseAuxList @
  [if <CondExp>                              ,  .< .~(aux <Args[DynArg->.<DynArg>.]>) >.
     then .< .~(<ThenBodyS[DynArg->.~DynArg]>) >.
     else .< .~(<ElseBodyS[DynArg->.~DynArg]>) >.]                                      )
```

```
              !isStatic(CondExp)       <CondBody> => (CondAuxList, <CondBodyS>)
   <ThenBody> => (ThenAuxList, <ThenBodyS>)        <ElseBody> => (ElseAuxList, <ElseBodyS>)
--------------------------------------------------------------------------------------------
                     if <CondExp> then <ThenBody> else <ElseBody>
                                        =>
(                                                 if .~(<CondExpS>)
     CondAuxList @ ThenAuxList @ ElseAuxList    ,   then .< .~(<ThenBodyS>) >.
                                                    else .< .~(<ElseBodyS>) >.    )
```

```
match exp with
  {pexp_desc = Pexp_ifthenelse(condExp, thenExp, elseExpOpt)} ->
    let (thenAux, thenExp') = buildStagedBody thenExp …
    in let (elseAux, elseExpOpt') =
      begin match elseExpOpt with
        None -> ([], None)
        | Some elseExp ->
            let (aux, e) = buildStagedBody elseExp …
            in (aux, Some e)
      end in
    let body = Exp.ifthenelse ~loc ~attrs
                 condExp thenExp' elseExpOpt'
    in (thenAux @ elseAux, body)
```

```
match exp with
  {pexp_desc = Pexp_ifthenelse(condExp, thenExp, elseExpOpt)} ->
      if isStaticExp condExp statVars
        then
          let auxName = fresh "aux" in
          let (auxAux, auxBody) = buildAuxBody …
          in let auxCall = buildAuxCall …
          in (auxAux @ [(auxName, auxBody)], auxCall)
        else
          let (condAux, condExp') = stage condExp in
          let (thenAux, thenExp') = stage thenExp in
          let (elseAux, elseExpOpt') =
            begin match elseExpOpt with
              None -> ([], None)
              | Some elseExp ->
                  let (aux, e) = stage elseExp in
                    (aux, Some (applyEsc e))
            end in
          let body = Exp.ifthenelse ~loc ~attrs (applyEsc condExp')
                       (applyEsc thenExp') elseExpOpt'
    in (condAux @ thenAux @ elseAux, applyBracket body)
```

Source Code Preprocessing

- Adding MetaOCaml Constructs
  - `.< … >.` `->` `[@metaocaml.bracket]`
  - `.~` `->` `[@metaocaml.escape]`
  - `!.` `->` `Pexp_apply ({pexp_desc =`

    ```
    Pexp_ident {txt = Lident “!."}},
    ...
    ```

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Results

```
let rec pow x n = if n = 0 then 1 else x * (pow x (n - 1))
[@@static []] [@@static [x]] [@@static [n]] [@@static [x; n]]
```

Staged MetaOCaml code:

```
let pow_ =
  .<let rec pow x n = if n = 0 then 1 else x * (pow x (n - 1))
    in pow  >.

let pow_x x =
  .<let rec pow n = if n = 0 then 1 else x * (pow (n - 1))
    in pow  >.

let pow_n n =
  let rec aux_8 x n =
    if n = 0 then .< 1  >. else .< .~x * .~(aux_8 x (n - 1))  >. in
  .< let pow x = .~(aux_8 .< x  >. n) in pow  >.

let pow_xn x n =
  let rec aux_7 x n =
    if n = 0 then .< 1  >. else .< x * .~(aux_7 x (n - 1))  >. in
.< let pow = .~(aux_7 x n) in pow  >.
```

Sample runs:

```
let _ = pow_x 3
- : (int -> int) code =
.<let rec pow_62 n_63 =
    if n_63 = 0 then 1 else 3 * (pow_62 (n_63 - 1)) in pow_62>.

let _ = pow_n 3
- : (int -> int) code =
.<let pow_65 x_64 = x_64 * (x_64 * (x_64 * 1)) in pow_65>.

let _ = pow_xn 3 3
- : int code = .<let pow_66 = 3 * (3 * (3 * 1)) in pow_66>.
```

Overview

- Automated Staging of OCaml Codes
  - Translation Rules
  - Source Code Annotation
  - Source Code Preprocessing
- Results
- Conclusions

Conclusions

- System that automates the staging process
  - Only by annotating the static information
    - No need to worry about staging constructs and levels
  - Get optimized code

Conclusions

- However…

```
let rec f x n = if n = 0 then 1 else x * (f (x+1) (n-1))
[@@static [x]]
```

```
let f_staged x =
  .< let f = if n = 0 then 1 else x * (f (n-1)) in f >.
```

?

Conclusions

- However…
  - Anonymous functions?
  - Curried functions?
  - Let-binding?

Conclusions

- However…

```
let plus_xn x n = .< let plus = x + n in plus  >.
# plus_xn 3 5;;
- : int code = .< let plus = 3 + 5 in plus>.
```

```
let plus_xn x n = let r = 3 + 5 in .< let plus = r in plus  >.
# plus_xn 3 5;;
- : int code = .< let plus = 8 in plus>.
```

Conclusions

- However…

```
.< let ff m =
     ((* CSP pow *) m) + ((* CSP pow *) m) + 0
   in ff >.
```

Conclusions

- System that automates the staging process
  - Only by annotating the static information
    - No need to worry about staging constructs and levels
  - Get optimized code
- However the success is still limited
  - There are various limitations which warrants further research

Any Questions?

Appendices

- Omitted slides for reference

Translation Rules

- Function with recursion
  - `let rec pow x n = if n = 0 then 1 else x * pow x (n - 1)`
  - If x is static

```
=> let rec pow_staged x =
     .< let pow n = if n = 0 then 1 else x * pow (n - 1) in pow >.

# pow_staged 5;;
- : (int -> int) code =
    .< let pow n = if n = 0 then 1 else 5 * pow (n - 1) in pow >.
```

Translation Rules

- Function with recursion
  - `let rec pow x n = if n = 0 then 1 else x * pow x (n - 1)`
  - If n is static

```
=> let rec pow_staged n =
     let rec aux x n = if n = 0 then .< 1 >. else .< .~x * .~(aux x (n-1)) >.
     in .< let pow x = .~(aux .<x>. n) in pow >.

# pow_staged 3;;
- : (int -> int) code = .< let pow x = (x * (x * (x * 1))) in pow >.
```

Translation Rules

For recursive function call in auxiliary function

```
isRecursiveCall()   isInAux()

<Args> => (ArgsAuxList, <ArgsS>)

<ArgsS'> = <ArgsS[DynArgS->.<DynArgS>.]>
--------------------------------------------------------------
<FunName> <Args>

=>

( ArgsAuxList, .< <FirstAuxName> .~(<ArgsS'>) >. ))
```

Translation Rules

For recursive function call in main function body

```
isRecursiveCall()   !isInAux()

<DynArgs> => (DynArgsAuxList, <DynArgsS>)

______________________________________

<FunName> <Args>

=>

( DynArgsAuxList, .< <FunName> .~(<DynArgsS>) >. )
```

Translation Rules

For all other function application

```
<Args> => (ArgsAuxList, <ArgsS>)
────────────────────────────────
<FunName> <Args>

=>

( ArgsAuxList, .< <FunName> .~(<ArgsS>) >. )
```

Translation Rules

- Function that uses other staged function
  - `let rec ff y m = if y = 0 then 0 else (pow y m) + (ff (y - 1) m)`
  - If y is static and pow staged over first argument

```
=> let ff_staged y =
     let rec aux y m =
       if y = 0 then .< 0 >. else .< (pow .~m) + .~(aux (y – 1) m) >.
     in .< let pow = .~(pow_staged y) in
     let ff m = .~(aux y .<m>.) in ff) >.
```

Translation Rules

For staged function application

```
isPrevStagedFun()

<DynArgs> => (DynArgsAuxList, <DynArgsS>)
──────────────────────────────────────────
<FunName> <Args>

=>

( DynArgsAuxList, .< <FunName> .~(<DynArgsS>) >. )
```

Results

```
let rec double xs = match xs with
                    | [] -> []
                    | x::xs -> (2 * x) :: (double xs)
[@@static []][@@static [xs]]
```

Staged MetaOCaml code:

```
let double_ =
  .<let rec double xs =
      match xs with | [] -> [] | x::xs -> (2*x)::(double xs) in
    double  >.

let double_xs xs =
  let rec aux_11 xs =
    match xs with
    | [] -> .< []  >.
    | x::xs -> .< (2 * x) :: .~(aux_11 xs)  >. in
  .< let double = .~(aux_11 xs) in double  >.
```

Sample runs:

```
let _ = double_xs [3;5;9]
- : int list code =
      .<let double_29 =
          [2 * 3; 2 * 5; 2 * 9]
        in double_29>.
```

Results

```
let nestedBranch x y =
  if x = 1 then (match y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy“) else "_"
  [@@static []][@@static [x]][@@static [y]][@@static [x; y]]
```

Staged MetaOCaml code:

```
let nestedBranch_ =
  .<let nestedBranch x y =
      if x = 1
      then match y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy"
      else "_" in nestedBranch  >.

let nestedBranch_x x =
  let rec aux_4 x y =
    if x = 1
    then .< match .~y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy"  >.
    else .< "_"  >. in
  .< let nestedBranch y = .~(aux_4 x .< y  >.) in nestedBranch  >.
```

```
let nestedBranch_y y =
  let rec aux_3 x y =
    match y with
     | 0 -> .< "x0"  >. | 1 -> .< "x1"  >. | _ -> .< "xy"  >. in
  .<let nestedBranch x =
      if x = 1 then .~(aux_3 .< x  >. y) else "_" in
    nestedBranch  >.

let nestedBranch_xy x y =
  let rec aux_2 x y =
    match y with | 0 -> .< "x0" >. | 1 -> .< "x1" >. | _ -> .< "xy" >.
  and aux_1 x y = if x = 1 then aux_2 x y else .< "_"  >. in
  .< let nestedBranch = .~(aux_1 x y) in nestedBranch  >.
```

Results

```
let nestedBranch x y =
  if x = 1 then (match y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy“) else "_"
  [@@static []][@@static [x]][@@static [y]][@@static [x; y]]
```

Sample runs:

```
let _ = nestedBranch_x 0
- : (int -> string) code =
.<let nestedBranch_5 y_4 = "_" in nestedBranch_5>.

let _ = nestedBranch_x 1
- : (int -> string) code =
.<let nestedBranch_7 y_6 =
    match y_6 with | 0 -> "x0" | 1 -> "x1" | _ -> "xy" in
  nestedBranch_7>.
```

```
let _ = nestedBranch_y 0
- : (int -> string) code =
.<let nestedBranch_9 x_8 =
    if x_8 = 1 then "x0" else "_" in nestedBranch_9>.

let _ = nestedBranch_y 5
- : (int -> string) code =
.<let nestedBranch_11 x_10 =
    if x_10 = 1 then "xy" else "_" in nestedBranch_11>.

let _ = nestedBranch_xy 1 2
- : string code = .<let nestedBranch_12 = "xy" in nestedBranch_12>.
```

Results

```
let rec ff y m =
  if y = 0 then 0 else ((pow y m)[@static.use ]) + (ff (y - 1) m)
[@@static []][@@static [y]][@@static [m]][@@static [y; m]]
```

Staged MetaOCaml code:

```
let ff_ =
  let pow = Runcode.run .< .~pow_  >. in
  .<let rec ff y m =
      if y = 0 then 0 else (pow y m) + (ff (y - 1) m) in
    ff  >.

let ff_y y =
  let pow = Runcode.run .< .~(pow_x y)  >. in
  let rec aux_21 y m =
    if y=0 then .< 0 >. else .< (pow .~m) + .~(aux_21 (y - 1) m) >. in
  .< let ff m = .~(aux_21 y .< m  >.) in ff  >.
```

```
let ff_m m =
  let pow = Runcode.run .< .~(pow_n m)  >. in
  .<let rec ff y = if y = 0 then 0 else (pow y) + (ff (y - 1)) in
    ff  >.

let ff_ym y m =
  let pow = Runcode.run .< .~(pow_xn y m)  >. in
  let rec aux_20 y m =
    if y = 0 then .< 0  >. else .< pow + .~(aux_20 (y - 1) m)  >. in
  .< let ff = .~(aux_20 y m) in ff  >.
```

Results

```
let rec ff y m =
  if y = 0 then 0 else ((pow y m)[@static.use ]) + (ff (y - 1) m)
[@@static []][@@static [y]][@@static [m]][@@static [y; m]]
```

<u>Sample runs:</u>

```
let _ = ff_y 3
- : (int -> int) code =
.<let ff_101 m_100 =
    ((* CSP pow *) m_100) +
      (((* CSP pow *) m_100) +
      (((* CSP pow *) m_100) + 0)) in
  ff_101>.
```

```
let _ = ff_m 3
- : (int -> int) code =
.<let rec ff_104 y_105 =
    if y_105 = 0
    then 0
    else ((* CSP pow *) y_105) + (ff_104 (y_105 - 1)) in
  ff_104>.

let _ = ff_ym 3 3
- : int code = .<let ff_107 = 27 + (27 + (27 + 0)) in ff_107>.
```