MetaOCaml Workshop ’04

Automatic Staging for Image Processing

Christoph A. Herrmann, Tobias Langhammer

University of Passau, Chair of Programming

Overview

- ***Introduction***
  implementation of image processing with staged execution
- ***Image Processing – Language Design***
  syntax, semantics, binding time analysis
- ***Example Image Filters***
  gradient filtering by convolution
- ***Image Processing – Language Implementation***
  datatypes, preprocessing, expression simplification, code generation with MetaOCaml
- ***Benchmark Results***
- ***Conclusions***

Introduction

Automatic Staging for Image Processing

Starting Point:

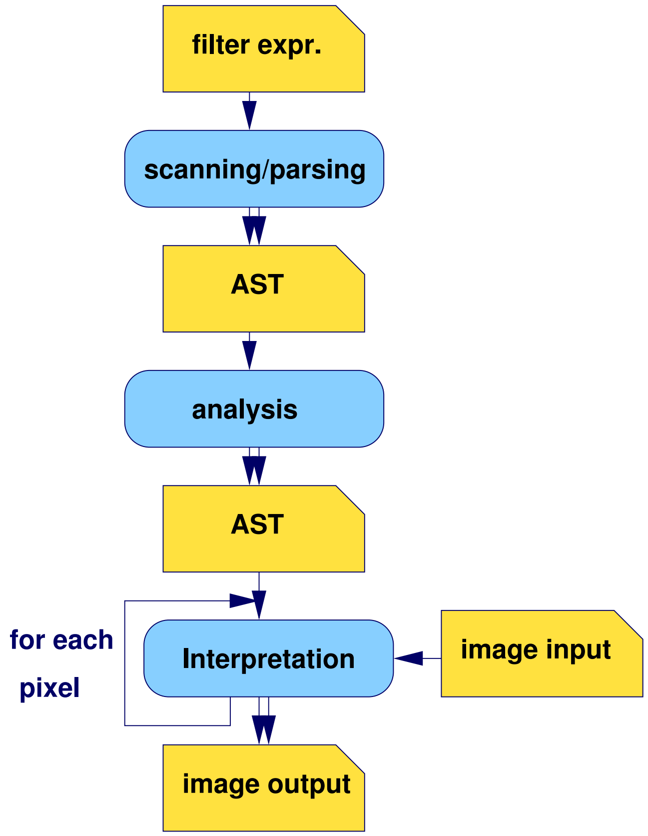
- Aim: rapid prototyping of filter expression language
- Domain of image processing needs fast execution.
- Interpretation too slow. No widespread tools for code generation.

Our Approach:

- MetaOCaml to eliminate overhead of interpretation
- Simplification phase determines code generation.
- Fast execution of residual program.

Image Processing by Interpretation

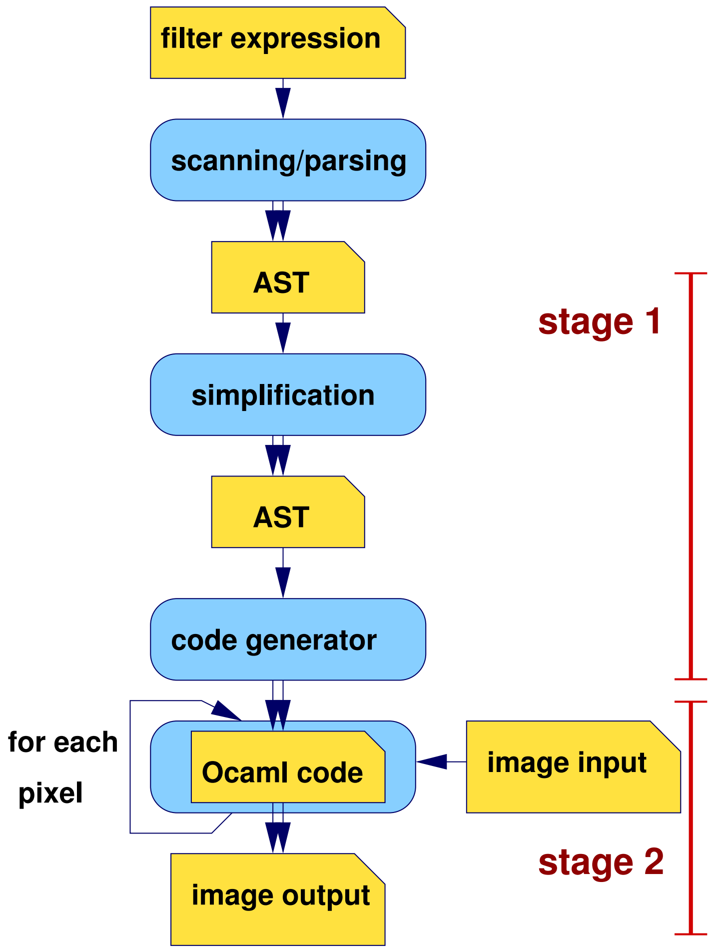
- *Filtering expression* as input
- *Scanning and parsing* to generate abstract syntax tree (AST)
- *Analysis* of AST (type-checking etc.)
- *Interpretation of AST* for each pixel of image
- Produces *filtered image as output*



Performance issue: *307200 interpretations for* 640×480 *pixel image !!!*

Compiling Image Filter Expressions

Interpretation replaced by *two stages*:



Stage 1: ***code generation***

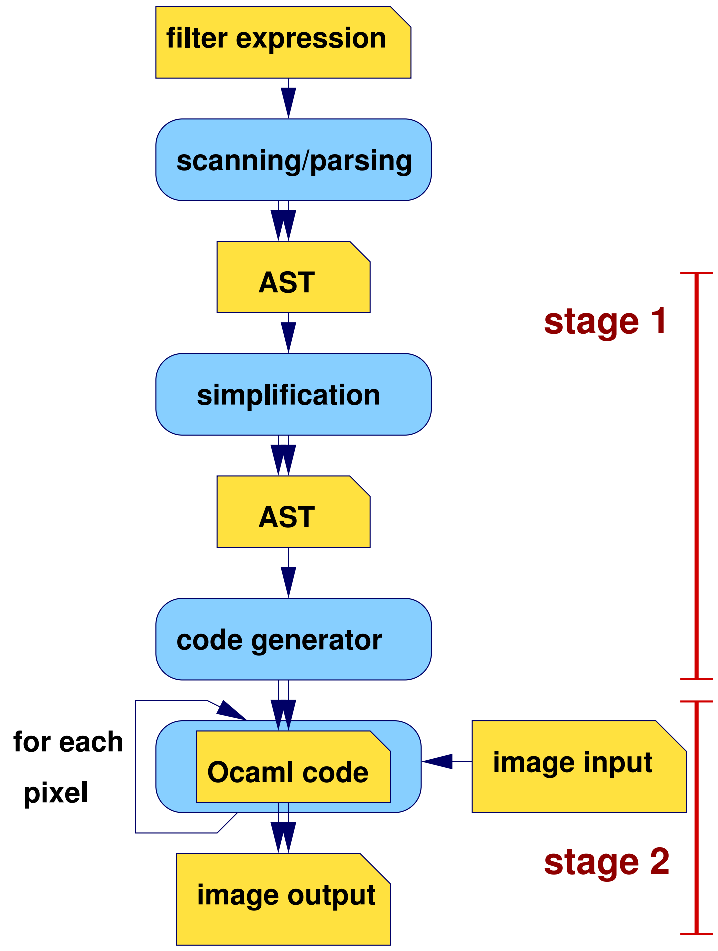
- **Simplification of AST**
- **Interpretation function automatically composed by staging annotations**
- **Generates residual program as code object**

Stage 2: ***filter application***

- **Runs residual program on each pixel.**

Compiling Image Filter Expressions

Interpretation replaced by *two phases*:



The two phases are semantically equivalent to interpretation!

Corresponds to first Futamura Projection

$!PE$ *interpreter source* $=$ *compiled_program*

Language Design Decisions

- Image values indexed by (row, col) coordinates for each color channel
- Filter defines new pixel with respect to its neighboring pixels
- Also non-local pixel access.
- Filtering language without binding time annotations.
  Advantages:
  - *the user is not bothered with binding time considerations*
  - *small changes may affect binding time of large code portions*
  - *simpler grammar*
  - *program analysis may be more sophisticated*
    Example: x dynamic, but x-x = 0 static
- Simplification by combined binding time analysis and static evaluation.
- Binding times:
  dynamic – expression dependends on `row` or `col`
  static – otherwise

Syntax of Image Filters

***A program consists of an expression*, which can be**

- **a *constant* (`42`, `true`, `3.14159`)**
- **a *variable* (`width`, `x`, `i`)**
- **a *parenthesized expression* or *operator* (`red(row,col)`, `max(0,b)`, `floor(3.5)`)**
- **a *conditional expression* (`if..then..else`)**
- **a *local definition* (`let..in..`)**
- **a *summation* (`sum..from..to..of..`)**

Semantics of Image Filters

- Language ***does not provide recursion***.
- Binary prefix operators `red`, `green` and `blue` for image access.
- int operators ***overloaded*** by float operators,
  coercion of int $\longrightarrow$ float,
  explicit conversion of float $\longrightarrow$ int (`floor`-operator).
- `let v=rhs in body`.
  Evaluates `rhs` and binds result to `v`. Evaluates `body` with new binding for `v`
- `sum i from a to b of body`.
  - Evaluate bounds `a` and `b`.
  - For each integral point within range: evaluate `body` with local variable `i` bound to current int
  - sum up evaluated bodies

Example

Gradient filtering by convolution:

```
0.5 + let m=[-1.0 0.0 1.0
            |-1.0 0.0 1.0
            |-1.0 0.0 1.0]
  in
  sum i from 0 to 2 of
    sum j from 0 to 2 of
      m[i,j] * red(row-i-1, col-j-1)
```

Examle: Convolution Filter

Gradient filtering by convolution:

Residual Program:

```
.<fun (row_1, col_2) ->
  let (c_3) =
   int_of_float ((0.5 +.
(((((-1. *. ((float_of_int rast.(row_1-0-1).(col_2-0-1).redChannel) /. 255.)) +.
     (0. *. ((float_of_int rast.(row_1-0-1).(col_2-1-1).redChannel) /. 255.))) +.
     (1. *. ((float_of_int rast.(row_1-0-1).(col_2-2-1).redChannel) /. 255.))) +.
  (((-1. *. ((float_of_int rast.(row_1-1-1).(col_2-0-1).redChannel) /. 255.)) +.
     (0. *. ((float_of_int rast.(row_1-1-1).(col_2-1-1).redChannel) /. 255.))) +.
     (1. *. ((float_of_int rast.(row_1-1-1).(col_2-2-1).redChannel) /. 255.)))) +.
  (((-1. *. ((float_of_int rast.(row_1-2-1).(col_2-0-1).redChannel) /. 255.)) +.
     (0. *. ((float_of_int rast.(row_1-2-1).(col_2-1-1).redChannel) /. 255.))) +.
     (1. *. ((float_of_int rast.(row_1-2-1).(col_2-2-1).redChannel) /. 255.)))))
     *. 255.) in
  if ((c_3) < 0) then 0 else if ((c_3) > 255) then 255 else (c_3)>.
```

Datatypes for Abstract Syntax Tree

```
type exp = Node of (dtype * op * exp list)

type dtype = Bool | Int | Float | Matrix

type op = C of value | V of string     | Read of color  | Int2Float
  | Floor            | UnOp  of unOp   | BinOp of binOp | If
  | Let of string    | IndexMatrix of string | Sum of string

type unOp = NegI | NegF | Not | ...

type binOp = AddI | SubI | MulI | ...

type color = Red | Green | Blue
```

Scanning and Parsing

- `ocamllex` generates scanner from token definition, defined by regular expression.
- `ocamlyacc` generates parser from context-free grammar + semantic actions.
- Semantic actions equipped with *type inference*.
- *Environment* to inherit type bindings.

Example rule

```
| LET VAR EQ baseExp IN baseExp
  {
    fun env ->
      let var,rhs = $2, $4 env in
      let env' = extEnv (var, dtypeof rhs) env in
      let body = $6 env' in
      Node (dtypeof body, Let var,[rhs;body])
  }
```

Binding Time Analysis and Static Evaluation

- Combined binding time analysis and static evaluation.
- Arguments: abstract syntax tree `expr` , environment `env` of static variables
- `expr` static ⇔ `expr` can be reduced at once ⇔ `unC expr` successfully yields a constant

A sample of the simplification function. . .

```
| Let s ->
    let [rhs;body] = args in
    let rhs' = subeval rhs in
    begin match unC rhs' with
    | Some v -> simplify body (extEnv (s,v) env)
    | None ->
        let body' = simplify body env in
        begin match unC body' with
        | Some v -> exp_of_value v
        | None -> exp_of_args [rhs';body']
        end
    end
```

Code Generation

Variant type for MetaOCaml code of each needed type

```
type 'a codevalue = CInt   of ('a,int)   code
                  | CFloat of ('a,float) code
                  | CBool  of ('a,bool)  code
```

A sample of the code generator function. . .

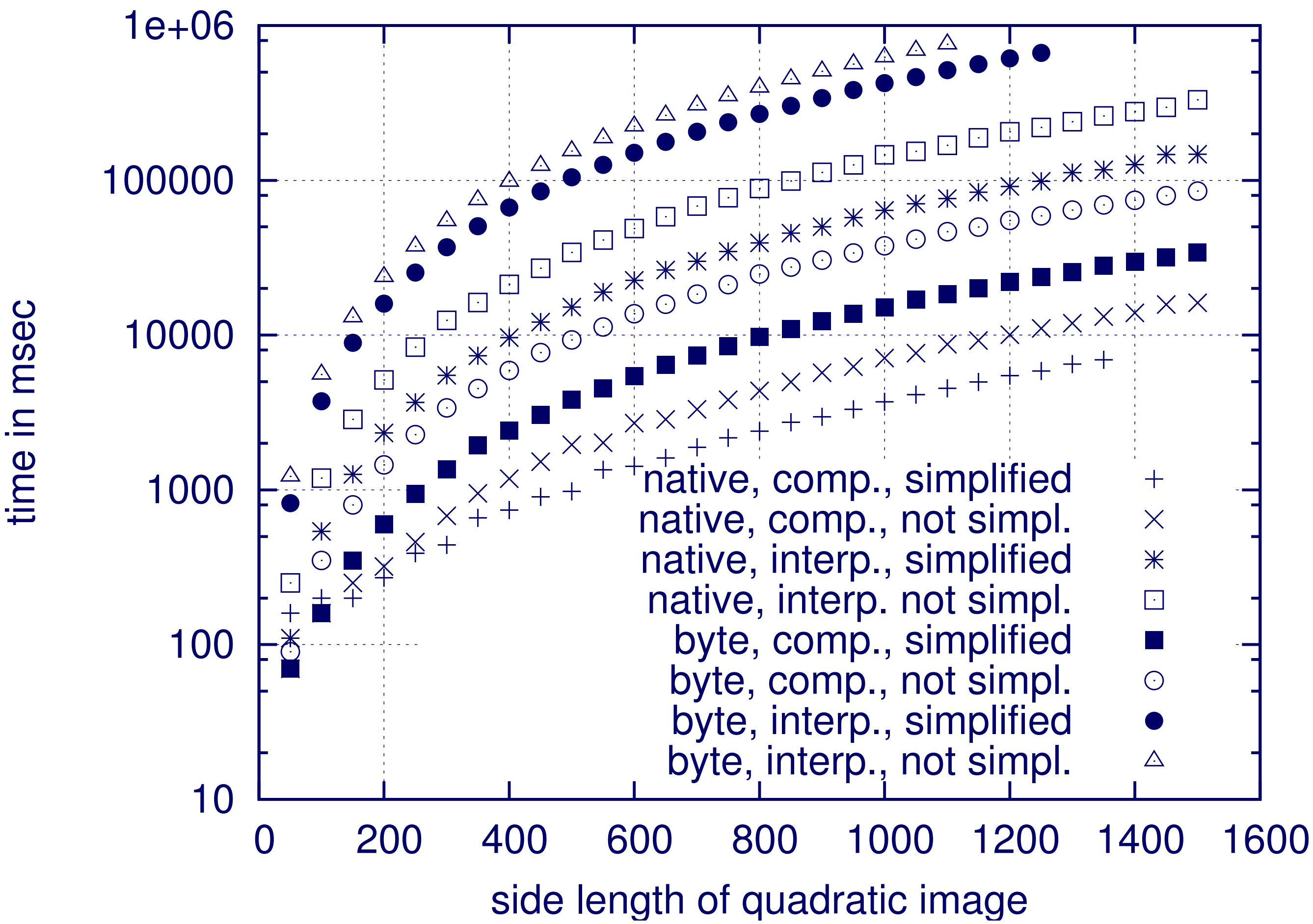
```
| Sum s ->
   let [lb;ub;body] = args in
   let CInt lb' = codegen lb env source
   and CInt ub' = codegen ub env source in
   let iteration i =
     let env' = extEnv (s,CInt i) env in
     codegen body env' source
   in
   let range = .<fromto .~lb' .~ub'>. in
   begin match dtype with
     | Int ->
         let addi = .<fun n i -> n + .~(unCInt (iteration .<i>.))>. in
         CInt .<List.fold_left .~addi 0 .~range>.
     | Float -> ...analogous...
   end
```

Bench-mark Results

filter	compilation	filter execution	simplifi-cation	time in msec.	time simplif. in msec.	speedup relative to ∗
gradient	byte code	compiled	yes	2863	0.2500	7.8708
			no	6663		3.3822
		interp.	yes	71783	0.2500	0.3140
			no	105873		0.2129
	native code	compiled	yes	880	0.0730	25.6098
			no	1437		15.6868
		interp.	yes	10780	0.0730	2.0906
			no	22537		1.0000 ⋆
zoom step fct.	byte code	compiled	yes	1187	0.0323	2.6713
			no	1223		2.5913
		interp.	yes	16567	0.0323	0.1913
			no	17600		0.1801
	native code	compiled	yes	490	0.0054	6.4694
			no	490		6.4694
		interp.	yes	2990	0.0054	1.0602
			no	3170		1.0000 ⋆
zoom interpol.	byte code	compiled	yes	2327	0.1237	7.0458
			no	2407		6.8116
		interp.	yes	68223		0.2403
			no	68903		0.2379
	native code	compiled	yes	750	0.0267	21.8578
			no	743		22.0538
		interp.	yes	16183		1.0130
			no	16393		1.0000 ⋆

Image size 640x480, Pentium III, 1GHz, 512MB

Timings for various image sizes



Conclusions and Future Work

- MetaOCaml useful for prototyping small domain-specific languages.
- Image filtering language w/o explicit binding time constructs.
- Automatic staging depending on analysis and simplification phase.
- Performance gain showed by benchmarks:
  - Staging and running bytecode faster than native-compiled interpreter.
  - Significantly good speedups for MetaOcaml with native code generation
- Looking forward to native-code compilation as part of MetaOCaml

Future Work

- Parallelization of image processing (OCaml binding to MPI)
- Higher degree of customisation (e.g. color intensities as int or float)

Thank you for your attention!