Octave Compiler

Rob Vermaas

Department of Information & Computing Sciences
Utrecht University,
The Netherlands

May 4, 2005

Octave

Language

- GNU Octave is a high-level language, primarily intended for numerical computations
- Open-source clone of MATLAB
- Interpreter
- Big set of (internal) library functions

Why?

- Heavily used for prototyping and simulations
  - Developers like the interactiveness of the interpreter
  - Only prototyping, real implementation in low-level language
- High-level language with high-level data structures
  - Programs are getting bigger
  - No need to worry about declarations
  - Easy, convenient syntax

But, this comes at a cost.

Why can Octave code be so slow?

Consider the following code,

```
b = c = zeros(10);

for x = 1:10
    a(x) = b(x)*c(x) ;
end
```

What happens at the assignment in the loop?

- Disambiguation
  - Distinguish function calls from array subscripts
- Type checking
  - Overloaded operators
- Bounds checking
- Reshaping

Octave compiler structure

- Octave Frontend
- Octave Optimizer
- Octave Type Inferencer
- Octave to C++ Backend

Octave Frontend

- Tools
  - Parsing and packing Octave code
  - Desugaring
  - Format checker
  - Pretty-printer
- Library
  - Generic Forward Propagation Strategy
  - Stratego-bindings for Octave interpreter-related functions
  - An SDF syntax definition for Octave

Parsing Octave

- No proper description of language
  - Manual dates from 1997!
- Old situation
  - Adaptation of parser of interpreter which needed complete evaluation
- New situation
  - Also based on interpreter
  - Bison/Flex based
  - Wrapper that outputs an ATerm
  - Foreign function interface (primitives)
- Have we abandoned SDF ?
  - No! We need SDF! We need concrete syntax!

Octave Optimizer

- Source-to-source data-flow transformations
- Classic approach
  - Build control flow graph and put in single-static-assignment form
  - Perform optimization
  - Convert back to 'normal' source code
- Our approach
  - Stay close to source, work on abstract syntax tree
  - Use generic data-flow-specification to perform optimizations
- Advantages
  - All information is available
  - Analysis and transformation in one step
  - Easy combination of optimizations
- Disadvantages
  - Optimizations using SSA can probably easier be reused for other languages (the optimization step on SSA-part)
  - Use general/known algorithms

Optimizations

Currently, octavec can perform the following optimizations,

- Constant folding
- Partial Evaluation
  - Constant Propagation
  - Copy Propagation
  - Common Subexpression Elimination
  - Function specialization (value-based)
- Dead Code Elimination
- Vectorization

Optimizations

Not much changed here, except rewriting the specifications to new style dataflow strategies.

- Prototyping in Tiger
- Easy transition from Tiger to Octave
- GFPS, Constant/Copy propagation, CSE within a day
- Basically only encode scopes of the language

Constant propagation

- Discover values that are constant on all possible executions of a program and to propagate these values through the program
- Best combined with constant folding and branch elimination

Constant propagation

- Discover values that are constant on all possible executions of a program and to propagate these values through the program
- Best combined with constant folding and branch elimination

```
x = 1

if x
  y = x+3
else
  y = 2
end

disp(y)
```

Constant propagation

- Discover values that are constant on all possible executions of a program and to propagate these values through the program
- Best combined with constant folding and branch elimination

```
x = 1

if x
  y = x+3
else
  y = 2
end

disp(y)
```

```
x = 1

y = 4

disp(4)
```

Constant propagation, implementation

```
strategies
  octave-prop-const =
    forward-prop(
      prop-const-transform(octave-prop-const)
    , prop-const-before(octave-prop-const)
    , octave-prop-const
    , prop-const-after
    | ["PropConst"], [], []
    )

  prop-const-transform(recur) =
       ElimIf; recur
    <+ ⟦ while <recur> do <id> end ⟧; ElimWhile; recur

  prop-const-before(recur) =
    ⟦ <id>[<*recur>] = <id> ⟧

  prop-const-after =
    prop-const-assign
     <+ PropConst  <+ EvalBinOp <+ EvalRelOp

  prop-const-assign =
    ?⟦ x = e ⟧
    ; where( <is-value> e )
    ; rules( PropConst.x : ⟦ x ⟧ -> ⟦ e ⟧ depends on [(x,x)] )
```

Partial evaluation

- Combination of
  - Value-based function specialization
  - Constant folding
  - Constant propagation
  - Copy propagation
  - Common subexpression elimination
- For each call, determine static arguments and specialize the called function for these arguments, and apply the given optimizations

Partial evaluation, implementation

```
strategies
  partial-eval =
    forward-prop(
        prop-const-transform(partial-eval) + pe-transform
      , prop-const-before(partial-eval)
      , partial-eval
      , pe-after
      | ["PropConst","CopyProp","CSE"], [], []
    )

  pe-transform =
    ?|[ f(a*) ]|
  ; specialize-call(specialize-function-by-value,specialization-facts-by-value | <get-nargout>)

  pe-after =
    try(copy-prop-after); try(prop-const-after); try(cse-after)
```

Partial evaluation, implementation

```
strategies
  partial-eval =
    forward-prop(
        prop-const-transform(partial-eval) + pe-transform
      , prop-const-before(partial-eval)
      , partial-eval
      , pe-after
      | ["PropConst","CopyProp","CSE"], [], []
    )

  pe-transform =
    ?[[ f(a*) ]]
  ; specialize-call(specialize-function-by-value,specialization-facts-by-value | <get-nargout>)

  pe-after =
    try(copy-prop-after); try(prop-const-after); try(cse-after)
```

Okay, cheated a bit by leaving out the function specialization part.

```
$ wc -l /pkg/src/octavec/octave-opt/eval/Octave-PE.str
78 /pkg/src/octavec/octave-opt/eval/Octave-PE.str
```

Why loop vectorization?

Remember,

```
b = c = zeros(10);

for x = 1:10
    a(x) = b(x)*c(x) ;
end
```

- Allows for parallelization
- Eliminates shape checks
- Eliminates reshapes
- Loop vectorization makes Octave code hard to read, therefore let compiler do it

Loop vectorization example

Original

```
function img = mono(r,g,b)
    [n,m] = size(r);
    img = zeros(n,m);

    for i = 1:n
        for j = 1:m
            img(i,j) = 0.3*r(i,j) + 0.6*g(i,j) +0.1*b(i,j);
        end
    end
end
```

Loop vectorization example

Original

```
function img = mono(r,g,b)
    [n,m] = size(r);
    img = zeros(n,m);

    for i = 1:n
        for j = 1:m
            img(i,j) = 0.3*r(i,j) + 0.6*g(i,j) +0.1*b(i,j);
        end
    end
end
```

Vectorized

```
function img = mono_vec(r,g,b)
    [n,m] = size(r);
    img = zeros(n,m);

    img(1:n,1:m) = 0.3*r(1:n,1:m) + 0.6*g(1:n,1:m) +0.1*b(1:n,1:m);
end
```

Typing Octave

- Type inferencing
  - Needed for efficient mapping in a backend
  - Possibilities for optimizations
  - Mainly a forward dataflow problem, therefore (again) GFPS
- Shape inference
  - Shape information is important for eliminating bound checks and reshaping
- Evaluating function calls and removing run-time checks

Typing Octave

- Type inferencing
  - Needed for efficient mapping in a backend
  - Possibilities for optimizations
  - Mainly a forward dataflow problem, therefore (again) GFPS
- Shape inference
  - Shape information is important for eliminating bound checks and reshaping
- Evaluating function calls and removing run-time checks

```
x = [ 1, 2, 3, 4]
[n, m] = size(x)
```

Typing Octave

- Type inferencing
  - Needed for efficient mapping in a backend
  - Possibilities for optimizations
  - Mainly a forward dataflow problem, therefore (again) GFPS
- Shape inference
  - Shape information is important for eliminating bound checks and reshaping
- Evaluating function calls and removing run-time checks

```
x = [ 1, 2, 3, 4]
[n, m] = size(x)
```

```
x = [ 1, 2, 3, 4]
n = 1
m = 4
```

Typing Octave

- Type inferencing
  - Needed for efficient mapping in a backend
  - Possibilities for optimizations
  - Mainly a forward dataflow problem, therefore (again) GFPS
- Shape inference
  - Shape information is important for eliminating bound checks and reshaping
- Evaluating function calls and removing run-time checks

```
x = [ 1, 2, 3, 4]
[n, m] = size(x)
```

```
x = [ 1, 2, 3, 4]
n = 1
m = 4
```

```
x = [1, 2, 3, 4]
if !ismatrix(x)
  printf("NO MATRIX!")
end
```

Typing Octave

- Type inferencing
  - Needed for efficient mapping in a backend
  - Possibilities for optimizations
  - Mainly a forward dataflow problem, therefore (again) GFPS
- Shape inference
  - Shape information is important for eliminating bound checks and reshaping
- Evaluating function calls and removing run-time checks

```
x = [ 1, 2, 3, 4]
[n, m] = size(x)
```

```
x = [ 1, 2, 3, 4]
n = 1
m = 4
```

```
x = [1, 2, 3, 4]
if !ismatrix(x)
  printf("NO MATRIX!")
end
```

```
x = [1, 2, 3, 4]
```

Type specification language / User input

User input

- Add type information for internal (non-user) octave functions
- Steer the type inference process
- Dynamic rules to interpret these rules

```
definitions
  constants
    pi      := float
    ...
  variables
    warn_fortran_indexing := int
    warn_num_to_str := int
    ...
  functions
    ...
    time    :: -> float
    fnmatch :: string, string -> int
    ...
```

- Embedded in Octave in Stratego

Octavec backend

- Generates C++ code
  - Stand-alone application
  - Dynamically loaded function (.oct)
- Links against liboctave
- Fallback to dynamic typing (liboctinterp)

Example (Dynamically typed, DLD)

```
function x = factorial(n)
     x = 1;
     while(n>1)
        x = x * n;
        n = n - 1;
     end
end
```

Example (Dynamically typed, DLD)

```
function x = factorial(n)
     x = 1;
     while(n>1)
        x = x * n;
        n = n - 1;
     end
end
```

Dynamically loaded function (to generate .oct file)

```
DEFUN_DLD (factorial, args, nargout, "")
{
  octave_value_list c_0;
  octave_value x;
  octave_value n;
  n = args(0);
  x = 1;
  while ( do_binary_op(octave_value::op_gt, n, 1).all().all().bool_array_value()(0) )
    {
      x = do_binary_op(octave_value::op_mul, x, n);
      n = do_binary_op(octave_value::op_sub, n, 1);
    }
  c_0(0) = x;
  return(c_0);
}
```

Example (Whole program compilation, TI)

```
function x = factorial(n)
     x = 1;
     while(n>1)
        x = x * n;
        n = n - 1;
     end
end
```

```
factorial(40)
```

Example (Whole program compilation, TI)

```
function x = factorial(n)
      x = 1;
      while(n>1)
          x = x * n;
          n = n - 1;
      end
end
```

```
factorial(40)
```

```
void d_0__a_0 ()
{
  double f;
  f = factorial__n_int(40);
}
double factorial__n_int (double n)
{
  double x;
  x = 1;
  while ( (n > 1) )
    {
      x = (x * n);
      n = (n - 1);
    }
  return(x);
}
```

Future work

a lot...