Octave (Matlab) compilation

Karina Olmos and Gordon Cichon

karina@cs.uu.nl, cichon@ifn.et.tu-dresden.de

Challenges for Digital Signal Processing

- High performance
  - Increasing algorithm complexity
- Low power
  - Weeks of Operation Time
  - Battery Power
- Development Time
  - Product cycle ~6 Months

The MOUSE project

Matlab **O**ptimized **U**niversal **S**ignal-Processing **E**nvironment

Goal: Provide an easily programmable platform

that combines:

- Flexibility of a FPGA
- Performance of an ASIC
- Cost and efficiency of an DSP

Why Octave?

To ease program development

- Heavily used for prototyping and simulations

Why Octave?

To ease program development

- Heavily used for prototyping and simulations
- High level language

Why Octave?

To ease program development

- Heavily used for prototyping and simulations
- High level language
- High level data structures (matrix)

Why Octave?

To ease program development

- Heavily used for prototyping and simulations
- High level language
- High level data structures (matrix)
- It is more declarative than procedural

Why Octave?

To ease program development

- Heavily used for prototyping and simulations
- High level language
- High level data structures (matrix)
- It is more declarative than procedural
- Rich library set
  - digital signal library
  - mathematical library
  - audio
  - video
  - ....

Octave challenges

- Undeclared, dynamically typed language

Octave challenges

- Undeclared, dynamically typed language
- Overloaded operators

  ```
  m = x * y
  ```

Octave challenges

- Undeclared, dynamically typed language
- Overloaded operators

  ```
  m = x * y
  ```
- Context dependent interpretation

  ```
  p = find([ 1 0; 0 1])                   p = [1 ; 4]
  [r, c] = find([ 1 0; 0 1])      ⟹     r = [1 ; 2] and c = [1 ; 2]
  [r, c, v] = find( [ 1 0; 0 1])          r = [1 ; 2] , c = [1 ; 2] and v = [1 ; 1]
  ```

Octave challenges

- Undeclared, dynamically typed language
- Overloaded operators

  `m = x * y`

- Context dependent interpretation

```
p = find([ 1 0; 0 1])                   p = [1 ; 4]
[r, c] = find([ 1 0; 0 1])      ⇒       r = [1 ; 2] and c = [1 ; 2]
[r, c, v] = find( [ 1 0; 0 1])          r = [1 ; 2] , c = [1 ; 2] and v = [1 ; 1]
```

- User defined functions
  - inherit overloaded features from operators
  - type and number of function arguments cannot be restricted
  - special type checks are introduced to prevent runtime-errors

Example

```
function retval = reshape (a, m, n)
  if (nargin == 2 && prod (size (m)) == 2)
    n = m(2);
    m = m(1);
    nargin = 3;
  endif

  if (nargin == 3)
    [nr, nc] = size (a);
    if (nr * nc == m * n)
       retval = zeros (m, n);
       if (isstr (a))
          retval = setstr (retval);
       endif
       retval(:) = a;
     else error('reshape: sizes must match');
     endif
    else usage('reshape(a, m, n) or reshape (a, size(b))');
  endif
endfunction
```

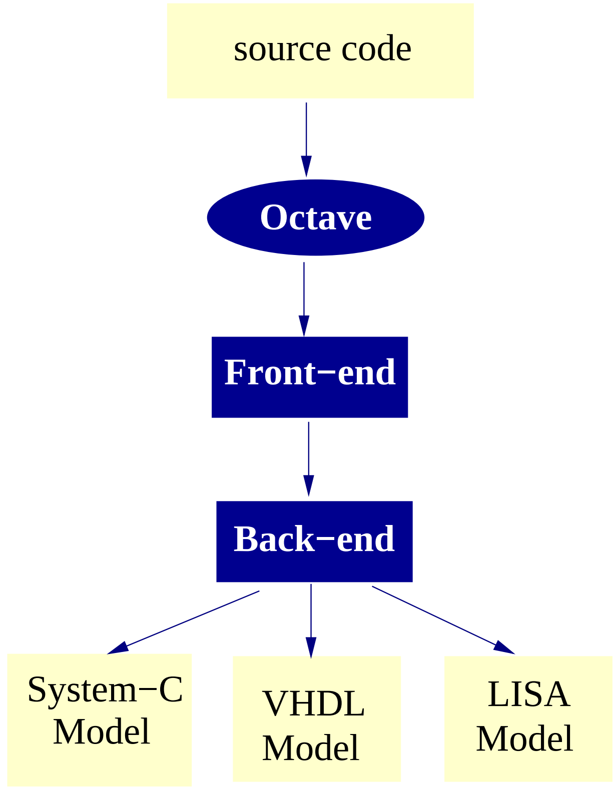
```
c = [ 1 2 3 ];
c1 = reshape(c, 3, 1);
```

Example ...

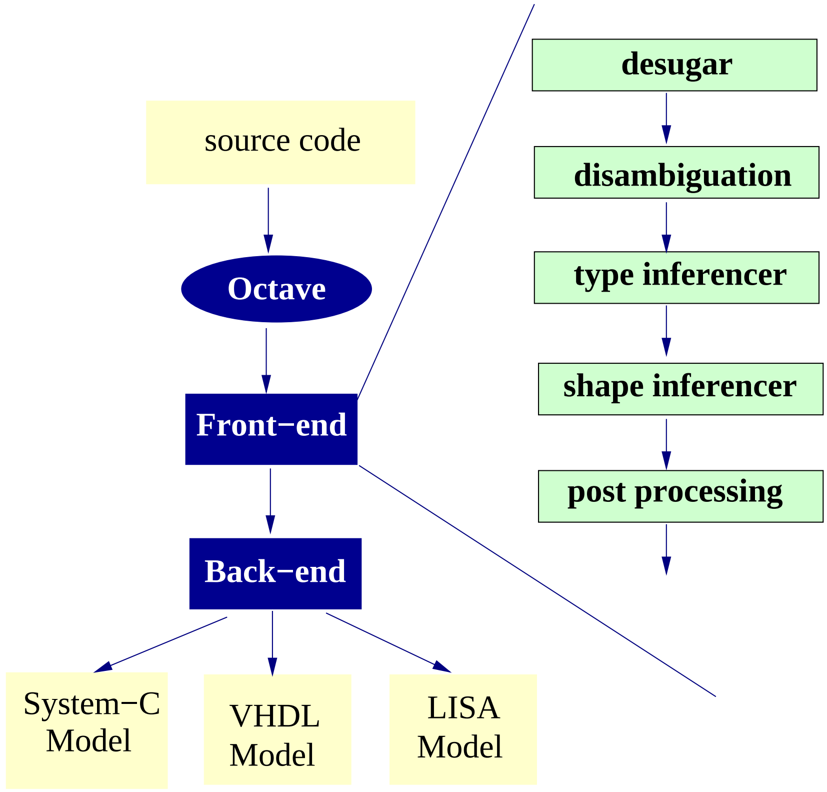
```
function retval = reshape_matrix(_int)_int_int(a, m, n)
   ()
   [nr, nc] = size_lib(a)
   if (nr * nc) == (m * n)
     retval = zeros_lib(m , n)
     ()
     retval(:) = a
   else
    error_lib('reshape: sizes must match');
   endif
end
```

```
 c = [ 1 2 3 ];
 c1 = reshape_matrix(_int)_int_int(c, 3, 1);
```

Structure of the compiler



Structure of the compiler



Desugaring

Complex statements are desugared into simpler ones

```
switch a
  case (1)
    x = 1;
  case (2)
    x = 2;
  case (3)
    x = 4;
  otherwise
    x = 5;
end
```

$\Rightarrow$

```
if a == 1
 then x = 1
 else
  if a == 2
    then x = 2
    else
     if a == 3
      then x = 4
      else x = 5
  endif
 endif
```

Desugaring

Complex statements are desugared into simpler ones

```
switch a
  case (1)
    x = 1;
  case (2)
    x = 2;
  case (3)
    x = 4;
  otherwise
    x = 5;
end
```

⇒

```
if a == 1
 then x = 1
 else
  if a == 2
    then x = 2
    else
     if a == 3
      then x = 4
      else x = 5
  endif
 endif
```

```
a = [10,11];
++a;
b = ++a ;
```

⇒

```
a = [ 10 11 ];
a = a + 1;
a = a + 1;
b = a;
```

Disambiguation

An identifier can represent a variable or a function depending on the context

```
function x = test(y)
   a = y + rand;
   rand(3) = a;
   x = rand(2) + 1;
```

$\Rightarrow$

```
function x = test(y)
  a = y + rand();
  rand[3] = a;
  x = rand[2] + 1;
end
```

Type inference

- No explicit type declarations
- Type casting is done transparently when required

Type inference

- No explicit type declarations
- Type casting is done transparently when required

```
function x = f(a, b)
  x = a + b;
```

```
c = 4;
r = f(c, c);
y = f([3, 4], r);
```

$\Rightarrow$

```
function x = f_matrix(_int)_int(a,b)
    x = b + a;
end
function x = f_int_int(a,b)
    x = a + b;
end
```

```
c = 4;
r = f_int_int(c, c);
y = f_matrix(_int)_int([3 4], r);
```

Shape inference

To determine the structure and size of a matrix

- Exact information avoids run-time memory allocation
- Give the same behaviour as the interpreter

Shape inference

To determine the structure and size of a matrix

- Exact information avoids run-time memory allocation
- Give the same behaviour as the interpreter

```
a = zeros(n);
b = zeros(n1);
......

m = a + b;
```

$\Rightarrow$

```
a = zeros(n);
b = zeros(n1)
  .....
m =
 if (n == n1)
   then a + b
   else error(...)
 endfi
```

Transformations

- Code hoisting
- Type and shape propagation
- Unreachable code elimination
- Program specialization
- Side effect elimination
- Delaying contextual information

Contributions

Classify intrinsic functions into groups

Func Group	Func Name	#In arguments	# Out arguments	Type
I	abs	fixed	fixed	Arg. dependent
II	sqrt	fixed	fixed	Value dependent
III	cumsum	1,2	fixed	First arg. dependent
IV	zeros	0,1,2	fixed	Static
V	size	1,2	Multiple	Context dependent

Contributions

Classify intrinsic functions into groups

Func Group	Func Name	#In arguments	# Out arguments	Type
I	abs	fixed	fixed	Arg. dependent
II	sqrt	fixed	fixed	Value dependent
III	cumsum	1,2	fixed	First arg. dependent
IV	zeros	0,1,2	fixed	Static
V	size	1,2	Multiple	Context dependent

```
[// fname,[arg-types], [ret-type]
 ("abs",   [INT],        S([INT]) ),
 ("abs",   [FLOAT],      S([FLOAT]) ),
 ("abs",   [COMPLEX],    S([FLOAT]) ),
 ("find",  [INT],        M([INT, [INT, INT],
                            [INT, INT, INT]])),
 ("find",  [FLOAT],      M([INT, [INT, INT],
                            [INT, INT, FLOAT]]))
]
```

Runtime elimination checks

Provide rules to eliminate unreachable code

```
EvalFirst:
  |[ &( (~int: 0){t}, e ) ]| ->
     |[ (~int: 0) {INT} ]|
EvalFirst:
  |[ &( e, (~int: 0){t} ) ]| ->
     |[ (~int: 0) {INT} ]|
EvalIfThen:
  |[ if (~int: 0){t} then e1 end ]| ->
     |[ nil {NIL} ]|
EvalFunc:
Call(Func("isstr"), [ e {STRING}] ) ->
  |[ (~int: 1) {INT} ]|
EvalFunc:
Call(Func("isstr"), [ e {t}] ) ->
  |[ (~int: 0) {INT} ]|
where <not(?STRING)>t
```

Including interpreter variables

Provide contextual information

```
tc-asg-multi(s) =
{| NumArgsOut:
   ?AssignMulti(vs, _)
   ; where(
         <length> vs => val
       ; rules(
           NumArgsOut: Var("nargout") -> val
         )
     )
   ; AssignMulti(id, s)
   ; try(TcAssg <+ TcAssg(s))
|}

tc-asg(s) =
{| NumArgsOut:
   Assign(id, id)
  ; rules(
      NumArgsOut: Var("nargout") -> 1
    )
  ; Assign(id, s)
  ; try(TcAssg <+ TcAssg(s))
|}
```

Conclusion and Future Work

- Static type determination
- Several compiler techniques were used for this purpose
- The framework eliminates most of the dynamic type checks

Conclusion and Future Work

- Static type determination
- Several compiler techniques were used for this purpose
- The framework eliminates most of the dynamic type checks
- Standard optimizations
- Vectorization
- Fixed-point data computation
- Translation into INSN (intermediate representation)