Table of Contents

1 Overview

1. Parsing
2. Resolve
3. Renaming and SSA
4. Type, rank, and shape inference
5. Constant propagation
6. Handle nargin and co.
7. Fix subscripts
8. Rewrite
9. Code Generation

1.1 Parsing

Matlab grammar is inherently ambiguous. The parser does not have enough information to tell between two possible meanings of the expression X(Y):

a. function call, where X is a name of a function and Y is the argument

b. array reference, where X is the name of the array and Y is the subscript.

The parser is responsible to convert syntactic shugar to built-in function calls, where conversion is unambiguous. This applies to [] constructs, transpose, and some others. For example, *foo(:)* is always an array reference, so it is translated by the parser as *foo[:]*.

1.2 Resolve

1.2.1 Phase 1

For each function, takes its parse tree and maps all variable references to sets of their possible definitions. Preserves the funcall/arrayref ambiguity.

1.2.2 Phase 2

For each function, resolve funcall/arrayref ambiguity as follows:

1. A funcall node, referencing a free name (using phase 1), which appears in the list of functions (closed-world assumption) is converted to either user-defined or built-in function call.
2. A funcall node, referencing a bound variable becomes an array ref.
3. A standalone unassigned variable reference becomes a function call (rand).

1.2.3 Remarks

1. Lambda functions and function pointers using @ are not, and will not, be supported.

1.3 Renaming and SSA

Renaming is immediate once defs sets are available after resolve-2, line numbers are embedded into the names. Phi nodes are generated for merges.

1.3.1 Closed World Assumption

We assume that all user-defined functions are passed to the compiler. Also, we have a list of all built-in functions.

1.4 Call Graph

Building call graph requires the ability to distinquish between function calls and array references (resolve-2). If G is not DAG, we drop enough back edges to convert G to DAG.

1.5 Constant propagation

Compile-time evaluation of type, rank, and shape is tightly connected with constant propagation.

TBD

1.6 Type, rank, and shape inference

In the following matlab to python example, we are forced to add code because we don't know type, rank, or shape of some variable.

```
x = 1:10 % matlab
```

```
x = arange(1,11).reshape(1,-1)  # python
```

Here, the right answer depends on what happens to *x*. If it's returned from the function, or its size is taken, reshape is required. Otherwise, reshape is not needed and may be harmful.

```
for k=1:size(L,1)-1
```

becomes

```
for k in arange(1,size(L,2)-1+1).reshape(1,-1).flat:
```

During type inference phase, we create a mapping from every Matlab function F---both built-in and user-defined---to function F*, so that F* takes the same number of arguments as F, but computes only the type and the shape of the result rather than the result itself. The body of F* is represented by Python class F.

For example, the python class node.transpose contains type inference information for matlab transpose operator:

```
# python
@extend(node.transpose)
def _type(self):
    """
    python class transpose represents matlab quote operator. The
    line below means that the type of the result is the same as
    the type of the first argument.
    """
    return self.args[0]._type()
```

Parse tree is processed in evaluation order. For assignments, leaf nodes are either constants, variable references, or formal parameters of user-defined functions. Inner nodes are built-in and user-defined function calls, possibly without arguments. The purpose of type inference is to represent types of all inner nodes as a function of type of formal arguments.

Normally, dependence are on variables already processed, so that their type is aready known. For loops, dependence may be on forward references. Solution: drop forward references ???

For built-in functions, it is easy to express that the type of the return value depends on the type of the argument:

Type of z depends on the type of the function argument t.

```
% matlab
function z=foo(t)
   z = t'
end
```

We create in compile-time a new python class foo. Then, we extend *foo* with a method _type

```
@extend(node.foo)
def _type(self):
    """
    foo is a user-defined matlab function, represented internally
    as class foo, created in compile time.  The line below means
    that type of the result is the same as the type of the first
```

```
    argument passed to foo.
    """
    return self.args[0]._type()
```

1.7 Fix subscripts

Increment or decrement array subscripts and subscript ranges, support boolean indexing and *end* keyword, resolve colon-ambiguity.

1.7.1 Colon-ambiguity

There is an ambiguity between colon as array subscript and colon as *range* function, which is resolved now.

1.7.2 Order of elements

Same in Fortran and Matlab. Not supported in Python. *Must detect somehow if order of elements is significant*

1.7.3 Upper and lower limit

Python

Unlike Matlab subscripts, which are one-based and include the upper limit, python subscripts are zero based and exclude the upper limit. For example, Matlab expression A(3:5) is converted to Python A[3-1:5-1+1]. Matlab indices 3:5 are [3 4 5]; Python indices [2:5] are [2,3,4]. All indices are decremented, then the upper bounds are incremented. If *step* is specified, it is moved to the end: A(3:4:5) is converted to A[2:5:4].

1.7.4 Boolean subscripts

Boolean subscripts are supported both by Matlab and by Python. In Fortran, boolean subscripts are expressed using *where* keyword. Boolean subscripts are detected by type inference, if available. Otherwise, if A, B, and C are expressions, and *op* is one of comparison operators, pattern A(B *op* C) is detected and marked as boolean indexing.

1.7.5 End-subscript

Python

Subscript *end* is translated to A.shape[B] where A is the array, and B is a zero-based dimension index.

Fortran

Subscript *end* is translated to SIZE(A,B) where A is the array, and B is a one-based dimension index.

1.8 Rewrite

1.9 Code Generation

These types are used to generate variable declarations during the code generation phase. We may need to duplicate the function body for every call site.

```
% matlab
function z=foo(t)
   z = t'
end

a = foo([1 2 3])
b = foo('hello')
```

Matlab function foo is translated to two f90 functions: foo_integer and foo_character.

```
! fortran90
CALL foo_integer(t,a)
CALL foo_character(t,b)
```

For merges, backward dependences should be consistent.

Presumably, in most cases the type of values returned by a builtin function call does not depend on the argument values, but only on their argument's type and shape. There are exceptions, such as *zeros*, whose type may be passed as argument.

Constant propagation should attempt to compute the type argument of zeros. If constant propagation is not available, the type argument may already be a constant. If unsuccessful, compilation is aborted or it should be specified explicitly. Finally, computation of the argument may be put off to run-time (python only).

```
% matlab
function z=foo(t)
    z = zeros([5 5],t)
end
```

May be converted as follows:

```
# python
def foo(t):
    if t=='ushort':
        return np.zeros([5,5],'uint16')
    if t=='single':
       return np.zeros([5,5],'float32')
```

Boolean indexing depends on type inference.

Finally, it may be possible to handle @ as function-handle type.

For non-recursive programs, user-defined function calls may appear only after these functions pass type inference.

1.10 Rewrite Phase

Constants are either numeric or character. Where possible, charachter constants are converted to python strings. Sometimes it is impossible:

```
m,n=size('hello world')
```

In such cases we must convert strings to np char arrays.

```
g=@(x) x**2
foo(g)

function foo(f)
foo = f(10)
end
```

We call programs that do not use @-expressions *@-safe*.

2 Debugging and other tricks

#.. image:: fig01.pdf

```
def my():
      print 8/2
```

Embedding dot graphs in latex:

```
sudo apt-get install dot2tex
pdflatex --shell-escape yourdoc.tex

http://www.texdev.net/2009/10/06/what-does-write18-mean/
http://www.duocoding.nl/blog/249/how-to-embed-dot-graphs-graphviz-in-latex
```

Debugging with gvim and Pyclewn:

```
Download pyclewn from github and install
:Pyclewn pdb main.py fastsolver.m
```

Debugging with emacs and pdb (linux only):

```
M-x pdb
Run pdb (like this): pdb main.py fastsolver.m -C dijkstra
(Pdb) b main.main
(Pdb) c
```

Setting breakpoints in node methods (_resolve, _type, _rank, _shape, _backend, _rewrite). For example, method foo is defined in file foo.py as follows:

```
@extend(node.foo)
def _foo(self):
    blah blah blah
```

Remember to import foo, or else class node does not have method _foo:

```
(Pdb) import foo
(Pdb) b node.function._foo
```

Same trick with .pdbrc:

```
!import node,type,rank,shape,resolve,rewrite,backend
b node.ident._type
```

Notes:

1. File *.pdbrc* is loaded first, before node, type, and the other modules are naturally imported. This is why trying to put a breakpoint in node.ident._type will fail unless we first !import both node and type.
2. Trailing newline is required in .pdbrc