Parametric Fortran - Automatic Program Generation for Scientific Computing

Martin Erwig
School of EECS
Oregon State University

This work is supported by NSF under grant ITR-0121542

Overview

Why Program Generation?

A Small Example

Parameter-Guided Program Generation

Parametric Fortran Features

Parametric Fortran 'Sociology'

Conclusions

OSU Oregon State University

Why Program Generation?

- Lack of abstractions makes code reuse difficult
- Copy&Paste is very error prone
- Maintenance of different code versions is expensive

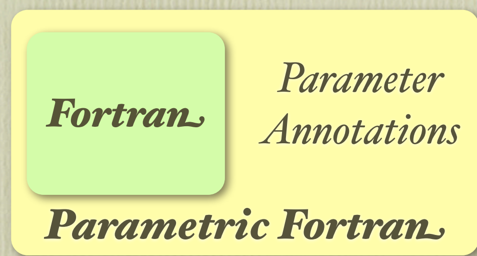
Oregon State UNIVERSITY

Concrete Motivation for Parametric Fortran

- Inverse Ocean Model (IOM) developed at OSU
- How to make it work for different ocean models?
- Write generic IOM system
- Abstract from model-specific data structures
- Capture model-specifics by parameters
- Generate customized versions of the IOM guided by parameter values

Oregon State UNIVERSITY

What is Parametric Fortran?



Parametric Fortran Program = Fortran Program Template

Parametric Fortran Example

Adding two arrays of arbitrary number of dimensions (assumption: size of each dimension is 1:100).

Number of dimensions for arrays

```
{dim:
  subroutine arrayAdd(a, b, c)
    real :: a, b, c
    c = a + b
  end subroutine arrayAdd
}
```

Generated Fortran

For dim=2:

```
subroutine arrayAdd(a, b, c)
  integer :: i1, i2
  real, dimension (1:100,1:100) :: a, b, c
  do i2 = 1, 100
    do i1 = 1, 100
      c(i1,i2) = a(i1,i2) + b(i1,i2)
    end do
  end do
end subroutine arrayAdd
```

Oregon State University

What Was Generated?

For dim=2:

New index variables → `integer :: i1, i2`

```
subroutine arrayAdd(a, b, c)
  integer :: i1, i2
  real, dimension (1:100,1:100) :: a, b, c
  do i2 = 1, 100
    do i1 = 1, 100
      c(i1,i2) = a(i1,i2) + b(i1,i2)
    end do
  end do
end subroutine arrayAdd
```

Oregon State University

What Was Generated?

For dim=2:

New index variables

Added Dimension

```
subroutine arrayAdd(a, b, c)
  integer :: i1, i2
  real, dimension (1:100,1:100) :: a, b, c
  do i2 = 1, 100
    do i1 = 1, 100
      c(i1,i2) = a(i1,i2) + b(i1,i2)
    end do
  end do
end subroutine arrayAdd
```

OSU Oregon State UNIVERSITY

What Was Generated?

For dim=2:

New index variables

Added Dimension

Added Loops

```
subroutine arrayAdd(a, b, c)
  integer :: i1, i2
  real, dimension (1:100,1:100) :: a, b, c
  do i2 = 1, 100
    do i1 = 1, 100
      c(i1,i2) = a(i1,i2) + b(i1,i2)
    end do
  end do
end subroutine arrayAdd
```

OSU Oregon State UNIVERSITY

What Was Generated?

For `dim=2`:

New index variables

Added Dimension

```
subroutine arrayAdd(a, b, c)
  integer :: i1, i2
  real, dimension (1:100,1:100) :: a, b, c
  do i2 = 1, 100
    do i1 = 1, 100
      c(i1,i2) = a(i1,i2) + b(i1,i2)
    end do
  end do
end subroutine arrayAdd
```

Added Loops

Added Array Indices

How Was It Generated?

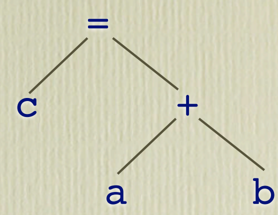
Parametric Fortran AST =
Fortran AST + parameter value at every node

- Traversal of Abstract Syntax Tree
- Applying "Parameter Effect" at every node

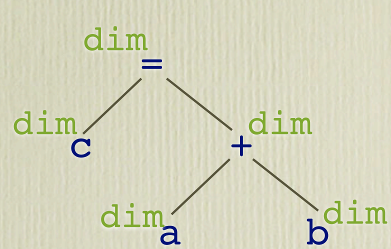
Transformation rules defined
for every affected syntactic category

Annotated Abstract Syntax Trees

`c = a + b` →

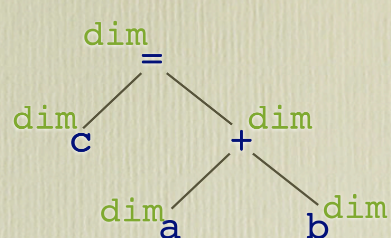


`{ dim: c = a + b }` →



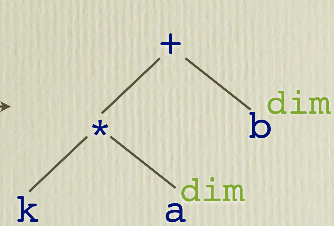
Annotated Abstract Syntax Trees

`{ dim: c = a + b }` →




```
k*{dim: a} + {dim: b}
{dim(a,b): k*a + b}
```

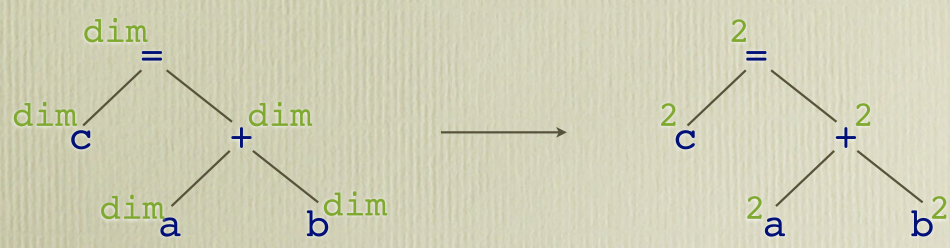
→



Program Generation: Step 1

```
arrayAdd.pf
  { dim: c = a + b }
```

```
pf {dim=2} arrayAdd.pf
```



Program Generation: Statements

Gen(dim=0, *stmt*) = *stmt*
Gen(dim=n, *stmt*) = do in=1,100 *Gen*(dim=n-1, *stmt*) end do

Gen(dim=2, c=a+b) = do i2=1,100 *Gen*(dim=1, c=a+b)
end do

```
= do i2=1,100
     do i1=1,100 Gen(dim=0, c=a+b)
     end do
  end do

= do i2=1,100
     do i1=1,100
        c=a+b
     end do
  end do
```

Program Generation: Variables

Gen(`dim=n`, *var*) = *var*`(i1,i2, ..., in)`

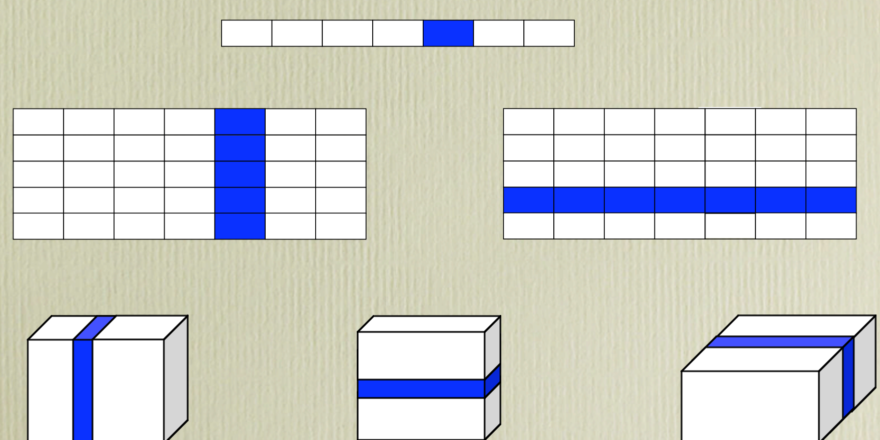
Gen(`dim=2`, `c`) = `c(i1,i2)`
Gen(`dim=2`, `a`) = `a(i1,i2)`
Gen(`dim=2`, `b`) = `b(i1,i2)`

... and so on for other syntactic categories

Oregon State University

Another Example: Array Slicing

There are *n* ways to slice an *n*-dimensional array on 1 dimension to obtain an (*n*-1)-dimensional array.



Oregon State University

Array Slicing

```
subroutine slice(a, k, b)
  {dim:   real :: a}
  {slice: real :: b}
  integer :: k
  {slice: b = a(k)}
end subroutine slice
```

Dimensions of input array

Dimensions of input array + dimension to slice on

```
dim=3
slice=(3,2)
```

```
subroutine slice(a, k, b)
  real, dimension (1:100,1:100,1:100) :: a
  real, dimension (1:100,1:100) :: b
  integer :: k
  integer :: i1, i2
  do i2 = 1, 100
    do i1 = 1, 100
      b(i1,i2) = a(i1,k,i2)
    end do
  end do
end subroutine slice
```


Array Slicing

More generally: project an *n*-dimensional array on *k* dimensions to obtain an (*n*-*k*)-dimensional array.

Parameter record

Parameter field (here: index vars)

Input dimensions

Output dimensions

outermost-only parameterization

```
subroutine slice(a, p.inds, b)
  {p.n: real :: a}
  {p.o: real :: b}
  integer :: p.inds
  {#p.o:
    {p.o: b} = {p: a(p.inds)}}
end subroutine slice
```

```
p = {n=4, o=2, dims=[1,3], inds=[i,j]}
```

Example parameter record

Generated Slicing Routine

```
p = {n=4, o=2, dims=[1,3], inds=[i,j]}

subroutine slice(a, i, j, b)
  real, dimension (1:100,1:100,1:100,1:100) :: a
  real, dimension (1:100,1:100) :: b
  integer :: i, j
  integer :: i1, i2
  do i2 = 1, 100
    do i1 = 1, 100
      b(i1,i2) = a(i,i1,j,i2)
    end do
  end do
end subroutine slice
```


Avoiding Code Duplication

```
program simulation
  {#stateVars:
    {stateVars.dim: real :: stateVars.name}}
  {#stateVars:
    {stateVars.dim: allocate(stateVars.name)}
    call readData(stateVars.name)
    call runComputation(stateVars.name)
    call writeResult(stateVars.name)
    deallocate(stateVars.name)}
end program
```

List parameter

```
stateVars = [temp, veloc]
temp  = {dim=3, name="temperature"}
veloc = {dim=2, name="velocity"}
```

Generated Simulation Program

```
program simulation
  real, dimension (:,:,:), allocatable :: temperature
  real, dimension (:,:), allocatable :: velocity
  allocate(temperature(1:100,1:100,1:100))
  call readData(temperature)
  call runComputation(temperature)
  call writeResult(temperature)
  deallocate(temperature)
  allocate(velocity(1:100,1:100))
  call readData(velocity)
  call runComputation(velocity)
  call writeResult(velocity)
  deallocate(velocity)
end program
```


Other Applications

- IOM tools:
  - Time convolution
  - Space convolution
  - Measurement modules
  - *many others ...*
- Automatic differentiation (generating tangent linear and adjoint models)

Automatic Differentiation

```
{diff:
  program model (x, y, …)
    …
    y = sin(x*x)
    …
  end program
}
```

Tangent linear model

```
diff = TL [x,y]
```

Active variables

```
program tl_model (x, y, tl_x, tl_y, …)
  …
  tl_y = 2*x*tl_x*cos(x*x)
  …
end program
```


Inviscid's Burger Model

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

$$u_0^1 = u_0^0 - \frac{\Delta t}{2\Delta x} u_0^0 (u_1^0 - u_X^0)$$

$$u_x^1 = u_x^0 - \frac{\Delta t}{2\Delta x} u_x^0 (u_{x+1}^0 - u_{x-1}^0), \text{ for } \; x = 1, 2, \ldots, X-1$$

$$u_X^1 = u_X^0 - \frac{\Delta t}{2\Delta x} u_X^0 (u_0^0 - u_{X-1}^0),$$

$$u_0^{t+1} = u_0^{t-1} - \frac{\Delta t}{\Delta x} u_0^t (u_1^t - u_X^t), \text{ for } \; t = 1, 2, \ldots, T-1$$

$$u_x^{t+1} = u_x^{t-1} - \frac{\Delta t}{\Delta x} u_x^t (u_{x+1}^t - u_{x-1}^t), \text{ for } \; t = 1, 2, \ldots, T-1 \quad x = 1, 2, \ldots, X-1$$

$$u_X^{t+1} = u_X^{t-1} - \frac{\Delta t}{\Delta x} u_X^t (u_0^t - u_{X-1}^t), \text{ for } \; t = 1, 2, \ldots, T-1$$

Generating Tangent Linear Model

```
{diff:
  subroutine burger(X,T,dx,dt,u)
    integer :: X, T
    real :: dx, dt, c
    real, dimension(0:X,0:T) :: u
    integer :: x, t, xm1, xp1, tm1, tp1
    c = dt / (2 * dx)
    do t = 0, T-1
      tp1 = t + 1
      tm1 = t - 1
      if (t == 0) then
        tm1 = 0
      else
        c = dt / dx
      end if
      do x = 0, X
        xp1 = x + 1
        xm1 = x - 1
        if (x == 0) then
          xm1 = X
        else if (x == X) then
          xp1 = 0
        end if
        u(x,tp1) = u(x,tm1)-u(x,t)*(u(xp1,t)-u(xm1,t))*c
      end do
    end do
  end subroutine burger }
```

Tangent linear model

Active variable

```
diff = TL [u]
```

```
program tl_burger (X,T,dx,dt,u,tl_u)
  …
  tl_u(x,tp1) = tl_u(x,tm1)-(tl_u(x,t)*
                (u(xp1,t)-(xm1,t))+
                u(x,t)*(tl_u(xp1,t)-
                tl_u(xm1,t)))*c
  …
end program
```


Generating Adjoint Model

```
{diff:
  subroutine burger(X,T,dx,dt,u)
    integer :: X, T
    real :: dx, dt, c
    real, dimension(0:X,0:T) :: u
    integer :: x, t, xm1, xp1, tm1, tp1
    c = dt / (2 * dx)
    do t = 0, T-1
      tp1 = t + 1
      tm1 = t - 1
      if (t == 0) then
        tm1 = 0
      else
        c = dt / dx
      end if
      do x = 0, X
        xp1 = x + 1
        xm1 = x - 1
        if (x == 0) then
          xm1 = X
        else if (x == X) then
          xp1 = 0
        end if
        u(x,tp1) = u(x,tm1)-u(x,t)*(u(xp1,t)-u(xm1,t))*c
      end do
    end do
  end subroutine burger }
```

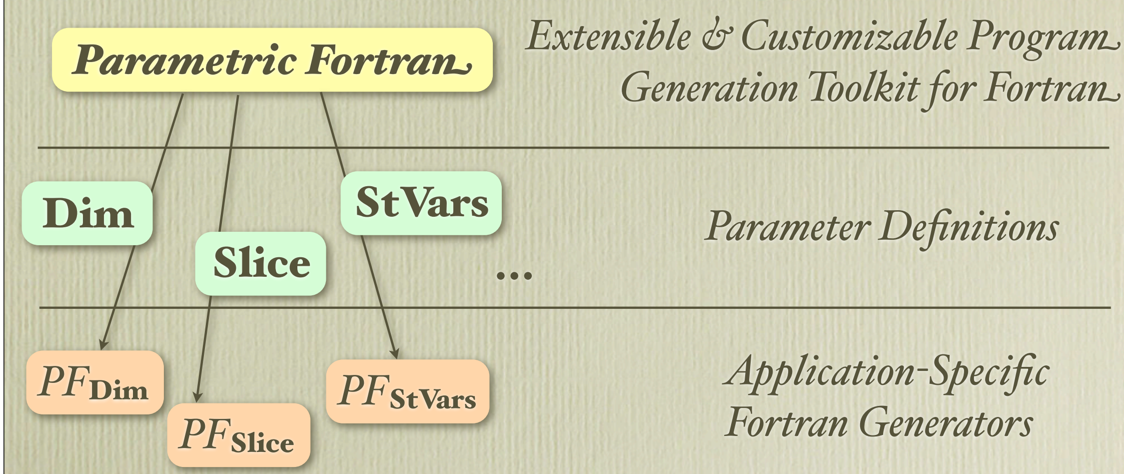
Adjoint model

Active variable

```
diff = AD [u]
```

```
program ad_burger (X,T,dx,dt,u,ad_u)
  …
  do t = T-1, 0, -1
    …
    do x = X, 0, -1
      …
      ad_u(x,tm1) = ad_u(x,tm1)+ad_u(x,tp1)
      ad_u(x,t)   = ad_u(x,t)-(c*(u(xp1,t)-
                    u(xm1,t)))*ad_u(x,tp1)
      ad_u(xp1,t) = ad_u(xp1,t)-(c*u(x,t))*ad_u(x,tp1)
      ad_u(xm1,t) = ad_u(xm1,t)+(c*u(x,t))*ad_u(x,tp1)
      ad_u(x,tp1) = 0
    end do
  end do
  …
end program
```

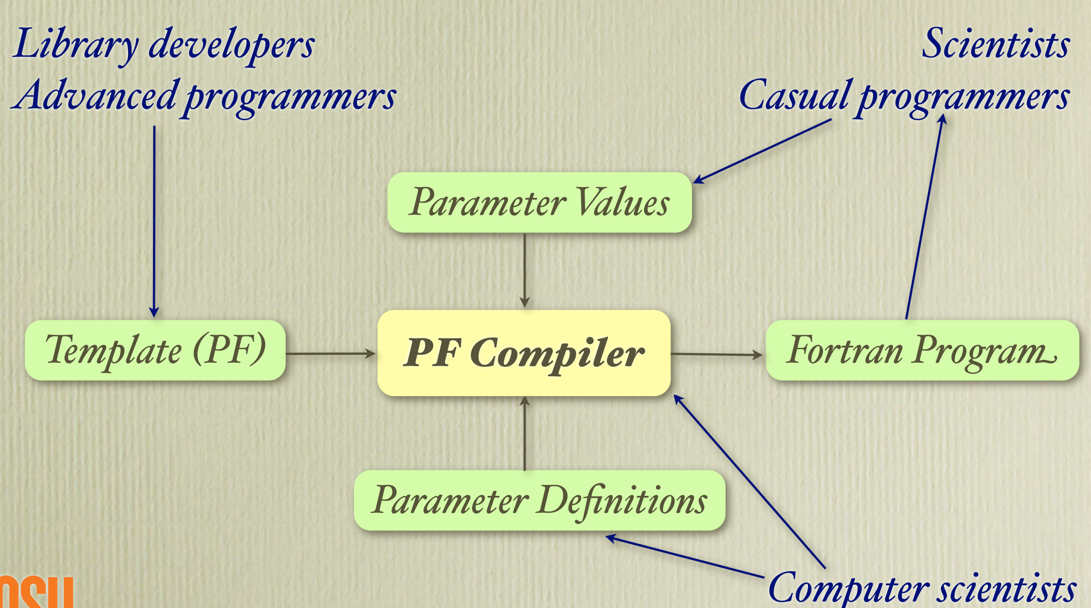
What is Parametric Fortran *Really*?



Parametric Fortran = *Fortran-Generator Generator*

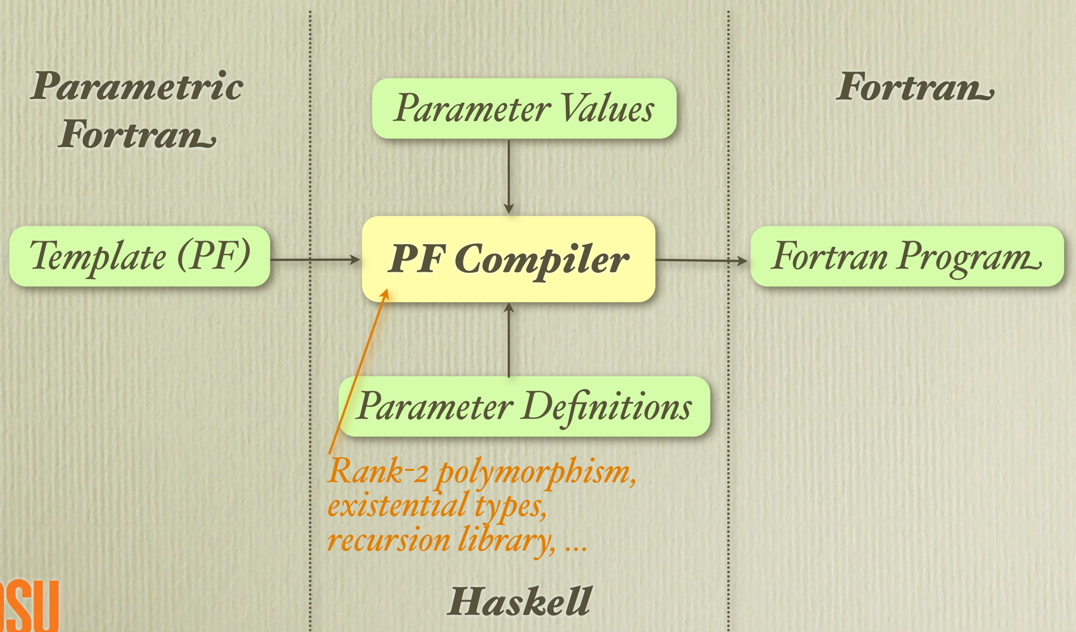
Oregon State University

Users & Collaboration



Oregon State University

Languages & System Architecture



Possible Future Work

- Define a type system for Parametric Fortran

```
p = {n=4, o=2, dims=[1,3], inds=[i,j]}

length dims = length inds
n = o + length dims
```

Allow specification of constraints, check constraints

- Parameter definitions for generating MPI code
- Domain-specific languages for program generation (in particular: DSELs)

Conclusions

- Parametric Fortran works well for the IOM (and beyond)
- Parametric Fortran is *not* type safe
- Domain-specific language might be better
- Scientists do use Fortran:
    - Reuse of existing Fortran code
    - Parametric Fortran is easier to ‘sell’ than Haskell

Conclusions

- *Program generation* is an *effective* approach to address software engineering problems in scientific computing
- *People* with different expertise (scientists, programmers, computer scientists) have to *collaborate*
- About *interdisciplinary work*:

  \- It is hard! (misunderstandings, frustration)

  \+ It is fun! (learn new things, cool things can happen)