ALPHA:

A functional data-parallel language based on Polyhedra

S. Rajopadhye

IRISA, Rennes

Outline

- Introduction
- ALPHA Syntax and Semantics (of Domains)
- Substitution & Normalization
- Change of Basis
- Conclusion

The Essence of Data Parallelism

- Collections of elementary data objects
- Pointwise Operations
- Alignment of different collections
- Conditional operations on (parts of) collections
- Reductions/Scans of associative and commutative operators

Alpha Syntax

```
system   ⟨Name⟩:     ⟨Domain⟩
                     ( [ ⟨Var⟩∗ : ⟨Domain⟩ of ⟨Type⟩ ]∗ )
           returns   ( [ ⟨Var⟩∗ : ⟨Domain⟩ of ⟨Type⟩ ]∗ );

var
                     [ ⟨Var⟩∗ : ⟨Domain⟩ of ⟨Type⟩ ]∗

let
       [ ⟨Var⟩= ⟨Expression⟩ ]∗
tel
```

Polyhedral Domains (Dual Representation)

- Integral Polyhedron: All points that **satisfy** a finite number of linear inequalities

$$\{z \in \mathcal{Z}^n \mid Qz \geq q\}$$

- Generators: All **convex combinations** of a set of vertices

$$\{z \in \mathcal{Z}^n \mid z = a^T G;\ \sum_i a_i = 1\}$$

- Alpha Syntax: $\{\langle\text{Idx}\rangle * \mid\ [\ \langle\text{IdExpr}\rangle \text{ >= } \langle\text{IdExpr}\rangle\ ]*\ \}$

Each Representation is useful in certain contexts. A standard (self dual) algorithm to convert from one to other.

Syntax of Expressions

- $\langle$Var$\rangle$| $\langle$Const$\rangle$ — Atomic
- $\langle$Expression$\rangle$ op $\langle$Expression$\rangle$ — Pointwise Ops
- $\langle$Domain$\rangle$: $\langle$Expression$\rangle$ — Restriction
- case [$\langle$Expression$\rangle$]$*$ esac — Case
- $\langle$Expression$\rangle$. $\langle$Dep$\rangle$ — Affine Dependency

where $\langle$Dep$\rangle$ has syntax: ($\langle$Idx$\rangle$$*$ -> $\langle$IdExpr$\rangle$$*$)

Example (Fibonacci)

```
system FibSys :  {N| 1<=N} () returns(F : integer );
var    Fib :     {i | 1 <= i <= N} of integer
let
 Fib =
       case
       {i| i<=2} :  1.(i -> ) ;
       {i| i>=3} :  Fib.(i -> i-1) + Fib.(i -> i-2) ;
       esac ;
 F =   Fib.( ->N) ;
tel ;
```

Semantics of Alpha Expressions

- Each expression denotes a function from a domain, D to a ⟨Type⟩ (real, integer or boolean)

- The Semantic function is compositional, and straightforward (eg, `A + B` denotes the sum of `A` and `B`) . . .

- Domains too are straightforward . . .

- except for affine dependency

Domains of ALPHA Expressions

- Constants and Variables $\boxed{Dom(c) = \mathcal{Z}^0}$ $\boxed{Dom(V)}$
- Pointwise operators $\boxed{Dom(e_1 \oplus e_2) = Dom(e_1) \cap Dom(e_2)}$
- Restriction $\boxed{Dom(D : e) = D \cap Dom(e)}$
- **case** operator $\boxed{Dom(\textbf{case}\ e_1; ..; e_n; \textbf{esac}) = \bigcup_{i=1}^{n} Dom(e_i)}$

Affine Depencency

- Q: What does the expression $e.(z \to f(z))$ mean?
- A: An expression whose value at z is the same as the value of e at $f(z)$

  Example: `Y = X.(i,j -> i)` or `Y[i,j] = X[i]`

- Its domain is $\boxed{Dom(e.f) = f^{-1}(Dom(e))}$

Substitution

```
substituteInDef[Fib, Fib, 1]; show[]
```

```
system FibSys :  {N| 1<=N} () returns(F : integer );
var    Fib :     {i | 1 <= i <= N} of integer
let
 Fib =
       case
       {i| i<=2} :  1.(i -> ) ;
       {i| i>=3} :    case
                      {i | i<=2} :  1.(i->);
                      {i | 3<=i} :  Fib.(i->i-1) + Fib.(i->i-2);
                      esac .(i -> i-1) + Fib.(i -> i-2) ;
       esac ;
 F =   Fib.( ->N) ;
tel ;
```

Normalization: syntax of a normal expression

- $\langle$BasicExpr$\rangle$ ::= $\langle$Var$\rangle$| $\langle$Const$\rangle$| $\langle$Var$\rangle$.$\langle$Dep$\rangle$ | $\langle$Const$\rangle$.$\langle$Dep$\rangle$

- $\langle$InnerExpr$\rangle$ ::= $\langle$BasicExpr$\rangle$ | $\langle$BasicExpr$\rangle$ op $\langle$InnerExpr$\rangle$
- $\langle$NormalExpr$\rangle$ ::=
  - $\langle$InnerExpr$\rangle$ | $\langle$Domain$\rangle$:$\langle$InnerExpr$\rangle$ |
  - case [$\langle$InnerExpr$\rangle$]$*$ esac
  - case [$\langle$Domain$\rangle$:$\langle$InnerExpr$\rangle$]$*$ esac

Normalization Rules (subset)

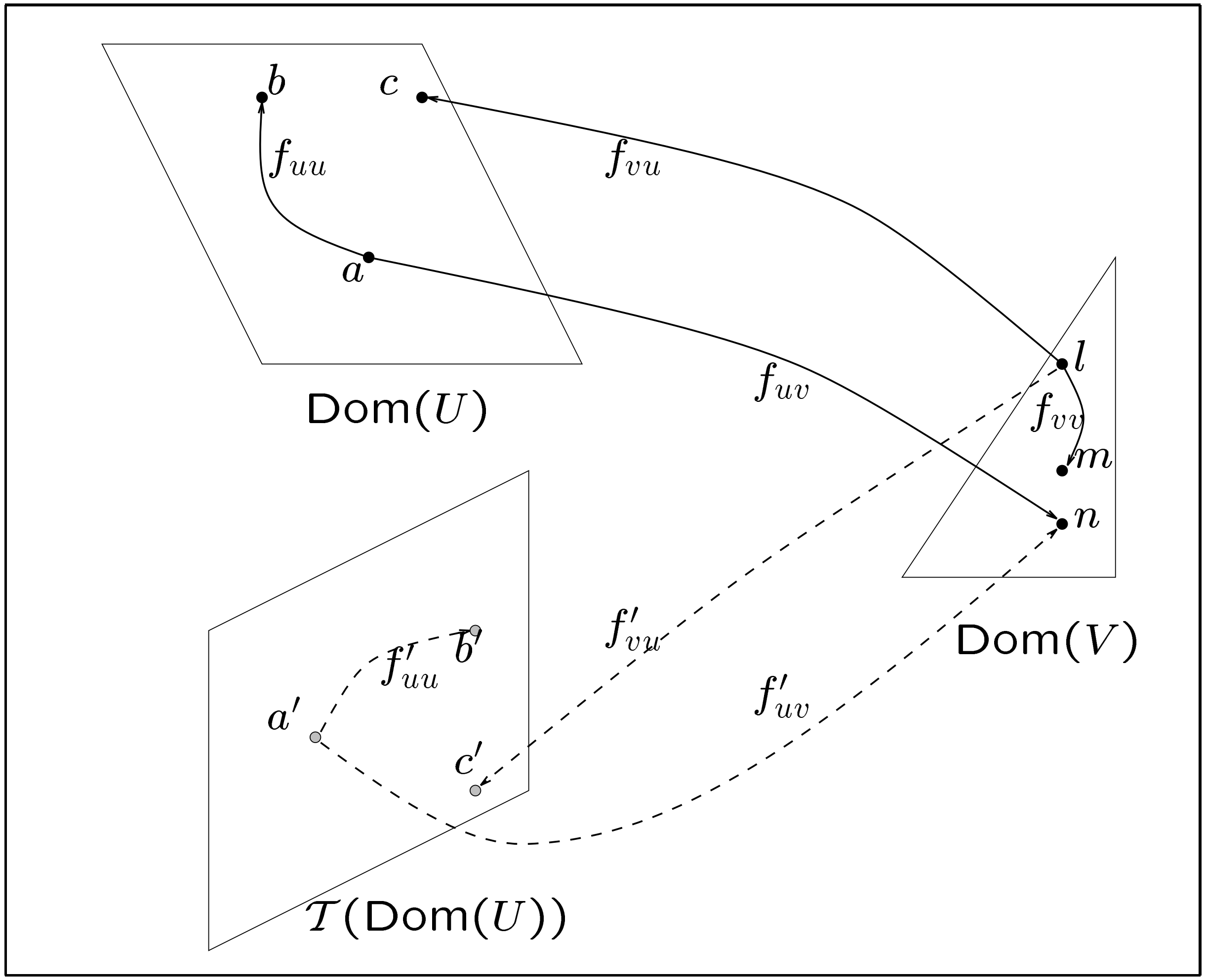
No.	Replace	By	Comments
1	$e.f$	e	if $f(z) = z$
2	$(e_1 \oplus e_2).f$	$(e_1.f) \oplus (e_2.f)$	
3	$e \oplus$ **case** $e_1;$ $\vdots$ $e_n;$ **esac**	**case** $e \oplus e_1;$ $\vdots$ $e \oplus e_n;$ **esac**	
4	$(D : e_1) \oplus e_2$	$D : (e_1 \oplus e_2)$	
5	$e_1 \oplus (D : e_2)$	$D : (e_1 \oplus e_2)$	
6	$(e.f_1).f_2$	$e.f$	$f = f_1 \circ f_2$
7	$D_1 : (D_2 : e)$	$D : E$	$D = D_1 \cap D_2$
8	$(D : e).f$	$D' : (e.f)$	$D' = f^{-1}(D)$

Normalization, example

```
normalize[]; show[]
```

```
system FibSys :  {N| 1<=N} () returns(F : integer );
var    Fib :     {i | 1 <= i <= N} of integer
let
 Fib =
       case
       {i| i<=2} :  1.(i -> ) ;
       {i| i=3}  :  1.(i ->) + Fib.(i -> i-2) ;
       {i| i>=4} :  Fib.(i -> i-2) + Fib.(i -> i-3)
                        + Fib.(i -> i-2) ;
       esac ;
 F =   Fib.( ->N) ;
tel ;
```

Change of Basis



Change of Basis Transformation Rules

- Space time transformations – applied to any (local) variable, V in an ALPHA system

- Unimodular transformation, T (specified like a dependency function)

- Three simple rewrite rules: Replace
  - the domain, D of V by $T(D)$
  - all occurrences of V on the rhs of all equations by $V.T$
  - E, the *entire rhs* of the equation defining V by $E.T^{-1}$

Example (Fibonacci again)

```
load["fib.a"]; changeOfBasis["Fib.(i -> -i)", "i"]; show[]
```

```
system FibSys :  {N| 1<=N} () returns(F : integer );
var    Fib :     {i | -N<=i<=-1} of integer
let
 Fib =
       case
       {i| i<=2} :  1.(i -> ) ;
       {i| i>=3} :  Fib.(i->-i) .(i -> i-1) + Fib.(i->-i) .(i -> i-2) ;
       esac . (i->-i) ;
 F =   Fib . (i->-i) . ( ->N) ;
tel ;
```

Normalize again

```
normalize[]; show[]
```

```
system FibSys :  {N| 1<=N} () returns(F : integer );
var    Fib :     {i | -N<=i<=-1} of integer
let
 Fib =
       case
       {i| -2<=i}:  1.(i -> ) ;
       {i| i<=-3}:  Fib.(i -> i+1) + Fib.(i -> i+2) ;
       esac ;
 F =   Fib.(->-N) ;
tel ;
```

Why does it work?

ALPHA domains (unions of finitely many polyhedra) constitute an abstract data type (ADT), closed under:

- Intersection
- Union (of finitely many members of the ADT)
- Preimage by (arbitrary) affine functions (the class of dependencies in ALPHA).
- Image by unimodular affine functions (the class of transformations used for changes of bases).