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
- Executing, Parallelizing and Compiling ALPHA

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: {⟨Idx⟩∗ | [⟨IdExpr⟩ >= ⟨IdExpr⟩]∗ }

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

Syntax of Expressions

- ⟨Var⟩| ⟨Const⟩ — Atomic
- ⟨Expression⟩op⟨Expression⟩ — Pointwise Ops
- ⟨Domain⟩ : ⟨Expression⟩ — Restriction
- case [⟨Expression⟩]∗ esac — Case
- ⟨Expression⟩. ⟨Dep⟩ — Affine Dependency

  where ⟨Dep⟩ has syntax: (⟨Idx⟩∗ -> ⟨IdExpr⟩∗)
- reduce (op, ⟨Dep⟩ , ⟨Expression⟩) — Reduction

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 and reductions

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 $e.(z \rightarrow f(z))$ 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 $\boxed{Dom(e.f) = f^{-1}(Dom(e))}$

Reductions

- Q: What does the expression $\texttt{reduce}(\oplus, (z \rightarrow f(z), e)$ mean?
- A1: Its domain is the projection of $Dom(z)$ by f:

$$\boxed{Dom[\texttt{reduce}(\oplus, (z \rightarrow f(z), e))] = f(Dom(e))}$$

- A2: Its value at z' is the application of $\oplus$ to the vlaues of e at all the points that are mapped to z' by f

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

- ⟨BasicExpr⟩::= ⟨Var⟩| ⟨Const⟩| ⟨Var⟩.⟨Dep⟩ | ⟨Const⟩.⟨Dep⟩
- ⟨InnerExpr⟩::= ⟨BasicExpr⟩| ⟨BasicExpr⟩op ⟨InnerExpr⟩
- ⟨NormalExpr⟩::=
  - ⟨InnerExpr⟩| ⟨Domain⟩ :⟨InnerExpr⟩|
  - case [⟨InnerExpr⟩]∗ esac
  - case [⟨Domain⟩ :⟨InnerExpr⟩]∗ esac

Normalization Rules (subset)

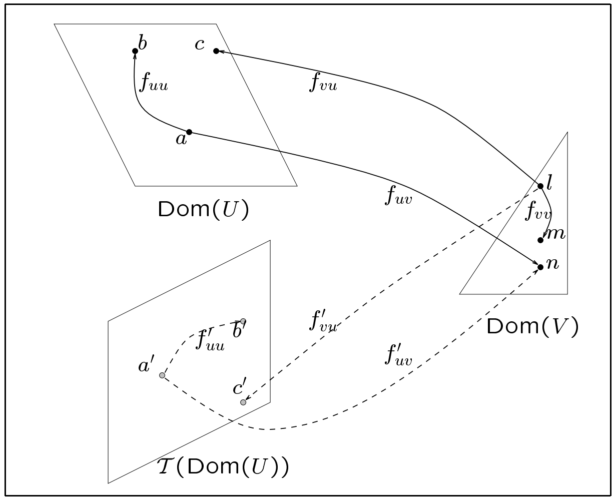
No.	Replace	By	Comments
1	$e.f$	e	if $f(z) = z$
2	$(e_1 \oplus e_2).f$	$(e_1.f) \oplus (e_2.f)$	
3	$e \oplus$ **case** e_1; ⋮ e_n; **esac**	**case** $e \oplus e_1$; ⋮ $e \oplus e_n$; **esac**	
4	$(D : e_1) \oplus e_2$	$D : (e_1 \oplus e_2)$	
5	$e_1 \oplus (D : e_2)$	$D : (e_1 \oplus e_2)$	
6	$(e.f_1).f_2$	$e.f$	$f = f_1 \circ f_2$
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).

Executing Alpha Programs

- (Horribly) Naive Interpretation
  - Just evaluate recursively according to natural semantics
  - Even `Fib` will take exponential time

Naive Interpretation

Memoize to avoid reevaluation: allocate storage for variable domains, and compute values as before:

- Visit all points in output var domains (polyhedron scanning)
- At each point compute the body of the equation as follows
  - Get arguments as follows
    - ∗ If not yet evaluated (indicated by a tag bit) ⇒ first suspend and (recursively) evaluate the argument.
    - ∗ If already evaluated, just read the value.
  - Compute equation body and store in table (and set tag).

Advantages:

- Works in the worst case (fall-back strategy).
- Improved complexity.

Drawbacks:

- Overhead of demand driven evaluation: context switching
- Memory inefficient
- Infinite domains: may not completely avoid reevaluation

Compilation

Avoid the drawbacks through static analysis:

- If a schedule exists ⇒ visit the points in that order ⇒ no suspensions (Wilde PhD '95: two to five fold improvement).
- Schedule ⇒ lifetime analysis ⇒ memory reuse (Quilleré PhD).
- Amenable to parallelization.

The Alpha Compiler: main steps

- Scheduling
- Alignment, Processor Allocation → COB
- Code Generation: Scanning unions of polyhedra
- Memory Reuse: Lifetime Analysis for validity of projections

Key Problem: Scheduling

Definition of schedule (timing function)

A function $t : \mathbf{V} \times \mathcal{D} \to \mathcal{N}$ such that whenever $U[x]$ depends on $V[y]$, then $t(U,x) < t(V,y)$. Domains may be arbitrarily large $\Rightarrow$ unreasonable to specify schedule by enumeration

- Affine schedules: Family of schhedules of the form

$$t(U,z) \equiv \lambda^U z + \alpha_U = \lambda_1^U z_1 + \lambda_2^U z_2 + \ldots \lambda_n^U z_n + \alpha^U$$

  for some row vector λ^U and scalar α^U

  Geometric interpretation: λ^U is a vector, normal to a family of hyperplanes. All points on the hyperplane $t(U,z) = \lambda^U z + \alpha_U$ are executed at time instant t, in parallel

Scheduling a (single) URE

Theorem 1: Given a URE $V[z] = f(V[z+d_1], \ldots, V[z+d_s])$ on a bounded domain, $\mathcal{D}$. $\lambda z + \alpha$ is a valid schedule iff for each of the s dependence vectors, d_k, and for all points $z \in \mathcal{D}$

$$\lambda z + \alpha > \lambda(z + d_k) + \alpha$$

i.e., $$d_k < 0$$

Finite number linear constraints (independent of the domain size). Scheduling $\equiv$ Linear Programming.

Geometric Interpretation: Find a family of hyperplanes, such that all dependence vectors point "backwards" (i.e., to a previous time instant).

Scheduling a SURE

- Use the same approach (a single schedule vector for all variables)
- Variable dependent schedules (schedule hyperplane for different variables have different slopes)
- Multidimensional schedules

Examples

Example 1

$$\begin{aligned} X[i,j] &= f(X[i-1,j+1]) && (1)\\ Y[i,j] &= g(X[i,j]) && (2) \end{aligned}$$

Example 2

$$\begin{aligned} X[i,j] &= f(X[i-1,j+1]) && (3)\\ Y[i,j] &= g(X[i-1,j+1]) && (4) \end{aligned}$$

More examples

Example 3

$$\begin{aligned} X[i,j] &= f(X[i-1,j+1]) && (5)\\ Y[i,j] &= g(Y[i+1,j-1]) && (6) \end{aligned}$$

Example 4

$$\begin{aligned} X[i,j] &= f(X[i-1,j+1]) && (7)\\ Y[i,j] &= g(Y[i+1,j-1],X[i,j]) && (8) \end{aligned}$$

still more examples

Example 5

$$X[i,j] = \begin{cases} \{\mid j=i=0\} & : \; 0\\ \{\mid i=0; j>0\} & : \; Y[i,j-1]\\ \{\mid i>0; j=0\} & : \; X[i-1,j+1]\\ \{\mid i,j>0\} & : \; f(X[i-1,j+1],Y[i,j-1]) \end{cases} \qquad (9)$$

$$Y[i,j] = \begin{cases} \{\mid j=0; i\geq 0\} & : \; X[i,j]\\ \{\mid j>0; i\geq 0\} & : \; g(X[i,j],Y[i+1,j-1]) \end{cases} \qquad (10)$$

Scheduling a (single) ARE

Given an ARE $V[z] = f(\ldots, V[A_k z + a_k]\ldots)$ on a bounded domain, $\mathcal{D}$. $\lambda z + \alpha$ is a valid schedule iff for each of the s dependence functions, $\langle A_k, a_k\rangle$, and for all points $z \in \mathcal{D}$

$$\lambda z + \alpha > \lambda(A_k z + a_k) + \alpha$$

Unfortunately, z does not "cancel out"

Thank God for Polyhedra

Theorem 2: Given an ARE $V[z] = f(\ldots, V[A_k z + a_k]\ldots)$ on a bounded polyhedral domain, $\mathcal{D}$. $\lambda z + \alpha$ is a valid schedule iff for each dependence function, $\langle A_k, a_k \rangle$, and for each vertex σ of $\mathcal{D}$,

$$\lambda\sigma > \lambda(A_k\sigma + a_k)$$

Retain the advantages: finite number linear constraints (independent of the domain size); same geometric interpretation.