A Retrospective of Crystal

Another "parallel compiler" ambition from the late 80's

Eva Burrows

BLDL-Talks
Department of Informatics, University of Bergen, Norway

February 16, 2010

WHAT IS CRYSTAL?

- a purely functional language with typed λ-abstraction, higher-order operators and data structures
- code resembles mathematical notations
- equational theory for program transformations (deriving new programs equivalent to the old) using a metalanguage
- efficient (parallel) compiler with optimizations (equations)
- all in all: addresses programmability and performance of parallel computations

CRYSTAL PROGRAMS

- a set of mutually recursive definitions of the form

```
<identifier> :  <type-expression> = <expression>
```

- with special constructs to express data parallelism and locality
- the compiler:
  - classifies the source code communication primitives and their cost on the target machine
  - maps data to distributed memory
  - generates parallel code with explicit communication commands

EXPRESSIONS

- basic data types: integers, booleans, floating points
- composite data types: index domains, data fields
- function application to constants and identifiers
- functions: $fn(x) : T\{\varepsilon[x]\}$
- environments: $\varepsilon\ where \{d_1, d_2, \ldots, d_n\}$
- conditionals: $\left\{ \begin{array}{c} b_1 \longrightarrow e_1 \\ \ldots \\ b_n \longrightarrow e_n \end{array} \right\}$
- reduction operators for binary functions: *reduce*, *scan*
- a construct to simulate while loops: $\mu(i) : D\{p[i]\}$, where $p[i]$ is a boolean predicate

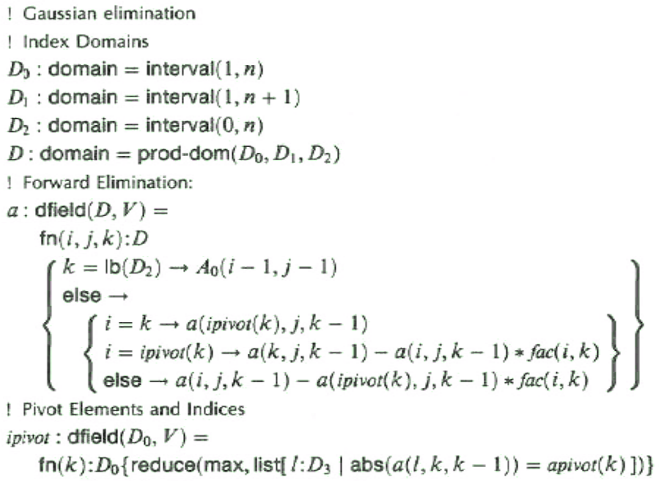
Composite Data Types: Index Domains

- an index domain *D* is associated with
    - set of elements (indices)
    - set of functions from *D* to *D* (communication operators) together with communication costs
    - set of predicates
- used to define distributed data structures
- they can be interpreted as locations in space and time over which the parallel computation is defined
- they can be declared to be of a certain *kind* (second-order types): universal, temporal, spatial, processor, memory (aid for the compiler)

INDEX DOMAINS (CONT.)

- basic index domains:
    - interval: $\mathit{interval}(m, n)$
    - hypercube: $\mathit{hcube}(n)$
    - binary tree: $\mathit{tree}(r, S)$ with root r and nodes from the set S (left or right associative or balanced)
- domain constructors: product ($D \times E$), disjoint union ($D + E$), function space ($D \rightarrow E$)
- data-dependent index domain: defined as the value of a recursively defined function
- hyper surfaces: define the boundary of domain where functions take no special value

EXAMPLE: FRAGMENT FROM THE GAUSS ELIMINTATION SOURCE CODE



! Gaussian elimination

! Index Domains

D_0 : domain = interval$(1, n)$

D_1 : domain = interval$(1, n+1)$

D_2 : domain = interval$(0, n)$

D : domain = prod-dom(D_0, D_1, D_2)

! Forward Elimination:

$$a : \text{dfield}(D, V) = \text{fn}(i, j, k){:}D \left\{ \begin{array}{l} k = \text{lb}(D_2) \rightarrow A_0(i-1, j-1) \\ \text{else} \rightarrow \\ \quad \left\{ \begin{array}{l} i = k \rightarrow a(ipivot(k), j, k-1) \\ i = ipivot(k) \rightarrow a(k, j, k-1) - a(i, j, k-1) * fac(i, k) \\ \text{else} \rightarrow a(i, j, k-1) - a(ipivot(k), j, k-1) * fac(i, k) \end{array} \right\} \end{array} \right\}$$

! Pivot Elements and Indices

$ipivot$: dfield(D_0, V) =

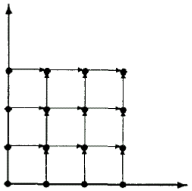
fn$(k){:}D_0\{$reduce(max, list$[\,l{:}D_3 \mid \text{abs}(a(l, k, k-1)) = apivot(k)\,])\}$

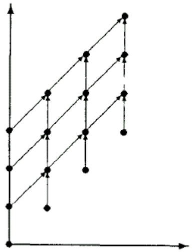
Index Domain Morphisms

- transform one index domain into an other, with the kinds of the domains indicating the change of interpretations
- a morphism is a mapping $D \longrightarrow E$ such that paths are preserved (paths as defined by the composition of the domains' communication operators)
- reshape morphisms: morphism with an inverse (isomorphisms)
- refinement morphisms: morphism with no inverse
- affine morphism is a reshape morphism from one product of intervals to an other (unify loop transformations, to "skew" the original structure):
  ex. loop interchange: $g = fn(i,j) : D_1 \times D_2\{(j,i) : D_2 \times D_1\}$
- space-time realizations by morphisms:
  $g = fn(i) : D\{(\varepsilon_s(i), \varepsilon_t(i)) : S[\mathscr{S}] \times T[\mathscr{T}]\}$
- can express inner loops to be executed in parallel
- etc.

Affine Domain Morphisms Example

- Let $D = 0..n-1$, $E = 0..n-1$, and $T = 0..2n-1$ be domains.
  Then, $g : D^2 \rightarrow E \times T$ s.t. $g = fn(i,j) : D^2.(i, i+j)$ is an affine morphism.
  The space-time realization of domain D requires a time product $D \times T$ while the reshaped domain $E \times T$ is a space time realization already.





Data Fields and Data Field Morphisms

- a data field is a function over an index domain into some domain of values: $a : D \longrightarrow V$, in Crystal notation: $a : dfield(D, V)$
- unify the notion of arrays and functions
- an example of a nested for-loop to compute the elements of a matrix:
  $$a : dfield(D^2, V) = fn(i,j) : D \left\{ \begin{array}{ccc} i = 0 \vee j = 0 & \longrightarrow & e_1 \\ else & \longrightarrow & a(i-1,j) + a(i,j-1) \end{array} \right\}$$
- index domain morphisms induce data field morphisms:
  Given $g : D \longrightarrow E$ index domain morphism, then
  $g^* : (D \longrightarrow V) \longrightarrow (E \longrightarrow V) : a \longrightarrow a \circ g^-1$
  will define to new data field. But only works if g is a reshape morphism

Communication Forms

- they needed the explicit introduction of senders, receivers (the sender does not know who the receivers are) to keep track of communications and help optimizations
- in the example: $a = fn(i) : D[\mathscr{P}]\{b(\tau_1(i)) + c(\tau_2(i))\}$
  the "request-reply" could be replaced by a single "send" if τ_1 or τ_2 (or both) had an inverse
- a *communication form* is a set of *communication actions* which are defined over two index domains D to E as a set of pairs: (s, t)
- the *invers* of the communication form is the set of pairs (t, s)

Compiler Directives

The compiler needs to be told about:

- domain morphisms: to run the program with reshaped data fields
- communication forms: to specify inverses that cannot be derived by the compiler
- common index domains: alignment of data fields to be allocated over the
- index precedence: $\mathit{precedence}(a, \{2, 1, 3\})$
- dynamic scheduling of mapping computations to processors. If not specified, it is static

The Equational Theory of Crystal

- a minimal set of valid equations/algebraic entities in the language ($M = N$) with some inference rules for deriving new equations from old
  - equations are expressing distributivity wrt. function composition and function application over conditional statements.
    Ex.:

$$\begin{aligned} F \circ \{B \rightarrow H\} &= \{B \rightarrow F \circ H\} \\ \{B \rightarrow H\} \circ F &= \{B \circ F \rightarrow H \circ F\} \\ &\vdots \end{aligned}$$

  - inference rules are the usual one for *equality*, *substitution*, *application*, *abstraction* and *composition*.
    Ex.:

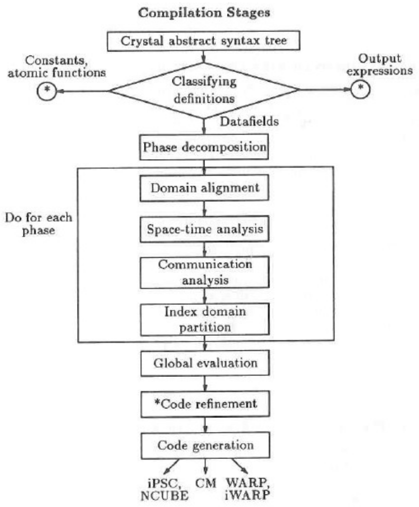
$$\frac{M = N}{fn(x)\{M\} = fn(x)\{N\}} \qquad \frac{M = N}{M[H/K] = N[H/K]} \qquad \cdots$$

- a *definition* is a simple equation where the left-hand side is a single variable, and the right-hand side an expression possibly containing the same variable

A New Program from the Old

- given $a : D \longrightarrow V$ a data-field, and $g : D \longrightarrow E$ index domain morphism, then we want a new program in terms of
  $a^* = a \circ g^- 1$
- how to do this without repeating the application of a and $g^- 1$ in the new program (since repeating them does not reduce the cost)?
- by a mechanizable strategy based
  - on substituting all occurrences of a with $a = a^* \circ g$ in the defintion of a
  - using a combination of unfolding g and $g^- 1$ and the identities of the underlying theory, eliminate all occurrences of g and $g^- 1$.
  - results in a new program (a more efficient one)
- a metalanguage is provided to define such (or similar) transformations at the programmer's convenience

THE STRUCTURE OF THE CRYSTAL COMPILER



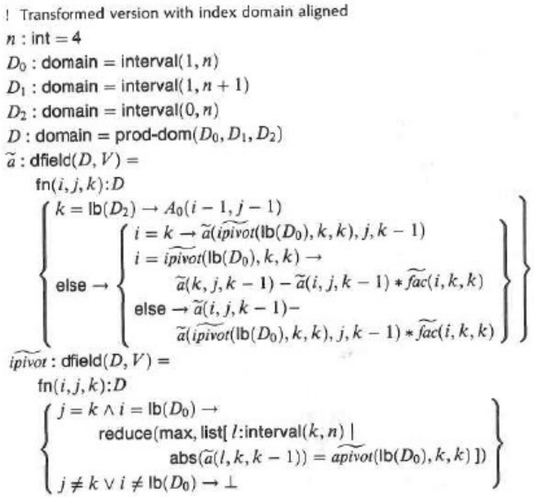
Dependency Analyses

- *call dependency* between data fields (functions). It helps decomposing the program into *phases* and determining the control flow of the target program
- *communication dependency* among the indices of the index domains (wrt to data fields defined over the same index domain). Provides information for data accessing patterns and storage information. Based on this the compiler determines appropriate reshape morphisms and mappings to processors and time sequences by other morphisms.

INDEX DOMAIN ALIGNMENT

- cross-referenced data fields: find a set of suitable reshape morphisms that map the index domains of these data fields into a common index domain
- data movements between processors are then reduced (optimization)
- can be automated or defined by the programmer

EXAMPLE: FRAGMENT OF THE TRANSFORMED GAUSS EL. CODE



! Transformed version with index domain aligned

$n : \text{int} = 4$

$D_0 : \text{domain} = \text{interval}(1, n)$

$D_1 : \text{domain} = \text{interval}(1, n + 1)$

$D_2 : \text{domain} = \text{interval}(0, n)$

$D : \text{domain} = \text{prod-dom}(D_0, D_1, D_2)$

$$
\begin{array}{l}
\widetilde{a} : \text{dfield}(D, V) = \\
\quad \text{fn}(i, j, k) : D \\
\quad \left\{ \begin{array}{l}
k = \text{lb}(D_2) \rightarrow A_0(i-1, j-1) \\
\text{else} \rightarrow \left\{ \begin{array}{l}
i = k \rightarrow \widetilde{a}(\widetilde{ipivot}(\text{lb}(D_0), k, k), j, k-1) \\
i = \widetilde{ipivot}(\text{lb}(D_0), k, k) \rightarrow \\
\qquad \widetilde{a}(k, j, k-1) - \widetilde{a}(i, j, k-1) * \widetilde{fac}(i, k, k) \\
\text{else} \rightarrow \widetilde{a}(i, j, k-1) - \\
\qquad \widetilde{a}(\widetilde{ipivot}(\text{lb}(D_0), k, k), j, k-1) * \widetilde{fac}(i, k, k)
\end{array} \right\}
\end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\widetilde{ipivot} : \text{dfield}(D, V) = \\
\quad \text{fn}(i, j, k) : D \\
\quad \left\{ \begin{array}{l}
j = k \wedge i = \text{lb}(D_0) \rightarrow \\
\qquad \text{reduce}(\max, \text{list}[\ l : \text{interval}(k, n) \mid \\
\qquad\qquad \text{abs}(\widetilde{a}(l, k, k-1)) = \widetilde{apivot}(\text{lb}(D_0), k, k)\ ]) \\
j \neq k \vee i \neq \text{lb}(D_0) \rightarrow \perp
\end{array} \right\}
\end{array}
$$

Synthesis of Parallel Control Structure

- determining the *temporal part* and the *spatial part* of an index domain from a communication dependency graph
- if the graph is acyclic, it is simple
- if it is cyclic: a sophisticated algorithm finds all the temporal components of the index domain, and the rest is spatial

Intermediate Program

- by now the Crystal program has been transformed into an intermediate “parallel” program for multiple processors with a global shared memory

DATA LAYOUT AND COMMUNICATION SYNTESIS

- deals with the distribution of data over individual memory on each processor, and
- defines explicit communications for the target program

REFERENCES

- Marina Chen, Young-il Choo, Jingke Li: Crystal: theory and pragmatics of generating efficient parallel code. In: *Parallel functional languages and compilers*, ed. B. Szymanski, ACM, 1991
- Marina Chen, Young-il Choo, Jingke Li: *Compiling parallel programs by optimizing performance*. The Journal of Supercomputing, 1988, Kluwer Academic Publishers, Boston
- J. Allan Yang, Young-il Choo: *Metalinguistic Features for Formal Parallel-Program Transformation*. Proceedings of the 1992 International Conference on Computer Languages, 1992.