Yale University
Department of Computer Science

Parallel-Program Transformation Using A Metalanguage

J. Allan Yang Young-il Choo

YALEU/DCS/TR-824
September 1990

This work has been supported in part by the Office of Naval Research under Contract No. N00014-90-J-1987.

Parallel-Program Transformation Using A Metalanguage

J. Allan Yang   Young-il Choo

Department of Computer Science
Yale University
New Haven, CT 06520-2158
yang@cs.yale.edu, choo@cs.yale.edu

September, 1990

Abstract

This paper describes how program transformation using a metalanguage can be an effective methodology for developing efficient parallel programs. As an example, a class of different parallel matrix multiplication programs are derived. Starting from a four-line, intuitive, though inefficient program, more sophisticated and efficient programs are derived. All the transformation steps preserve the semantics of the initial program so that the transformation process is a verification of the equivalence among all the derived programs. The metalanguage provides a simple, flexible, extensible, and formal framework for expressing transformational schemes. It also automates the cumbersome and error-prone part of the program transformation.

1 Introduction

Stepwise refinement has been widely recognized as an effective programming methodology. Programmers are advised to first come up with simple and high-level algorithms that are easy to understand and verify. Refinements are then applied to transform the high-level algorithms into more concrete programs. Similarly, program transformation can be an effective methodology for developing *efficient* programs. Since efficiency and readability are often conflicting goals, programmers are encouraged to begin with simple, intuitive, and easy to verify programs. Transformations are then applied to derive more efficient ones. Because manually transforming the programs is usually tedious and error-prone, program transformation systems have been built to automate this process for traditional sequential languages [9, 8, 2, 10, 11, 13, 1].

As various parallel machines have become available in the past few years, how to effectively program these machines becomes a pressing issue. Crystal [4] is a parallel language aimed at addressing this issue with a very high-level functional language and a smart compiler. Sophisticated parallelizing compilation technologies for Crystal have been devised for some commercially available parallel machines [5]. Though the parallelizing compiler can produce high quality parallel code for certain classes of programs, the problem of producing efficient parallel code for *all* source programs is intrinsically very difficult and sometimes intractable. There are many occasions when it is clear for programmers how to transform their source programs for better efficiency, yet it is difficult for the compiler to recognize these special cases without numerous *ad hoc* knowledge built into the compiler. Source-to-source program transformation has been proposed as a methodology for transforming programs which have irregular or nonlocal communication patterns into programs which have only uniform and localized communication [3]. Though the intuition behind the transformations is not too difficult, the technicalities required to crank out the new program are often quite involved and cumbersome. A metalanguage has been presented to automate the transformation in [6]. As a result, Meta-Crystal [14] is designed to provide programmers with a flexible, extensible, and formal tool for implementing their transformational schemes as meta procedures for use on similar occasions. Meta-Crystal is a language with Crystal programs, definitions, and expressions as fundamental objects and is equipped with quoting, unquoting, constructors, selectors, predicates, and semantic-preserving operators for constructing and manipulating pieces of Crystal text.

This paper describes how program transformation using a metalanguage such as Meta-Crystal can be an effective methodology for developing efficient parallel programs. As an example, a class of different parallel matrix multiplication programs are derived starting from a four-line intuitive though inefficient program. A more sophisticated and efficient program is derived at the end. This paper is organized as follows. We introduce a few basic notations and terminologies in the paragraph below, as they will be used through out this paper. Section 2 briefly introduces the simple syntax of Crystal and Meta-Crystal, how Crystal programs are interpreted on parallel machines, and how Crystal programs are manipulated in Meta-Crystal. The core of this paper is in section 3 where we demonstrate how a class of matrix multiplication programs can be derived using Meta-Crystal. Section 4 provides some concluding remarks.

Brief Notes on λ-Notation

Because λ-notation is used in both Crystal and Meta-Crystal for function declarations and some properties of λ-calculus [7] are used in Meta-Crystal for transforming programs, we provide a brief introduction to the λ-notations in this paragraph. The λ-notation gives us a way to express a function without giving an explicit name. For example, $\lambda x.x+1$ is an anonymous function that takes an argument x (i.e., argument is written after λ) and returns a value $x+1$ (i.e., the function body is written after the "." and extends to the right as far as possible). If we want to give $\lambda x.x+1$ a name f, we write it as $f = \lambda x.x+1$. The traditional way is to write it as $f(x) = x+1$. We write $(\lambda x.x+1)(3)$ to mean $\lambda x.x+1$ is applied to 3. That is, we simply concatenate the function and the argument. We call an expression like $\lambda x.x+1$ a *λ-abstraction* term, and expressions like $(\lambda x.x+1)(3)$ or $f(3)$ *application* terms. A *β-redex* is an application term with the rator (i.e., the function) being a λ-abstraction term, e.g., $(\lambda x.x+1)y$ is a β-redex. To *β-reduce* a β-redex is to carry out the application of that redex, e.g., β-reducing $(\lambda x.x+1)(y)$ will result in $y+1$. Let f be any function, doing *η-abstraction* on f results in an equivalent function $\lambda x.(f(x))$. That is, f is equivalent to $\lambda x.(f(x))$ by η-abstraction.

2 Crystal and Meta-Crystal

We first briefly introduce the abstract syntax of both Crystal and Meta-Crystal. Then we describe, very roughly, how Crystal programs are interpreted on parallel machines. Finally, we describe how Crystal programs are manipulated in Meta-Crystal.

2.1 Abstract Syntax

Both Crystal and Meta-Crystal are lexically scoped functional languages based on a simple syntax. The syntax is basically the notation for the λ-abstraction and application from λ-calculus [7], enriched with conditional, recursion, and local definitions. Its style is similar to many modern functional languages such as ML [12]. We describe the common syntax of both languages briefly below, assuming readers are familiar with this style of languages that the descriptions below are explanatory. Let $x, x_1, x_2, \ldots$ range over identifiers, $e, e_1, e_2, g_1, g_2, \ldots$ range over expressions:

- A *program* consists of a set of mutually recursive definitions with an output expression, having a form as below:

$$x_1\texttt{=}e_1\texttt{,}\ x_2\texttt{=}e_2\texttt{,}\ \ldots\texttt{,}\ \texttt{?}e\ .$$

  The notation x_1=e_1 is a *definition* that binds the identifier x_1 to the value of e_1. The result of the program is the value of the output expression ?e. Since the evaluation is driven by need, the order of definitions is irrelevant.

- *Local* definitions can be introduced to an expression with the where{...} construct:

$$e\ \texttt{where\{}\ x_1\texttt{=}e_1\texttt{,}\ x_2\texttt{=}e_2\texttt{,}\ \ldots\texttt{\}}\ .$$

  Local definitions can be mutually recursive and may have their own local definitions.

- The *conditional expressions* have the following form:

$$\left\{\begin{array}{l} g_1 \to e_1, \\ g_2 \to e_2, \\ \ldots \end{array}\right\}$$

  The expressions $g_1, g_2, \ldots$ are guards for their associated expressions and should be evaluated to boolean values. The value of a conditional expression is the first expression with a true guard.

- *Functions* are introduced by the λ-abstraction and have the form $\lambda(x_1, x_2, \ldots).e$ where $x_1, x_2, \ldots$ are the formal arguments and e is the body of the function. Function applications have the form of $e_1(e_2, e_3, \ldots)$, where e_1 is the function and $e_2, e_2, \ldots$ are the arguments. When there is only one formal argument, we may omit the parenthesis around it.

- *Typed expressions* have the form $e_1 : e_2$, where e_2 should be evaluated to a type. The formal of a function abstraction can be typed as well: $\lambda x\texttt{:}e_1\texttt{.}e$.

- An *expression* can be a constant (of type integer, float, etc.), an identifier, a conditional expression, a function abstraction, a function application, or a typed expression.

Parenthesis can be used for grouping. All non-prefixed notations are considered as the sugared form of their equivalent prefixed function applications. For example, `1+2` is the sugared form of `add(1,2)`.

2.2 Parallel Interpretation of Crystal Programs

Due to space limitation, we only provide a *very* rough sketch to how Crystal programs are interpreted on parallel machines so as to help readers understand the examples presented in the next section. Readers are referred to [3, 5, 6] for more details.

The major objects of interest in Crystal are data fields and index domains. A *data field* is a function over some *index domain*. An *index domain* is a space of index points. For example, "D = interval(1,n)" is an interval index domain with n index points, indexed from 1 to n. More complicated index domains can be constructed by cartesian product. For example, $D \times D$ (or D^2 for short) is an index domain with n^2 index points, indexed from $(1,1)$ to (n,n). Data fields over index domains are to be distributed among processors and computed in parallel. Since data field definitions can be mutually recursive, *data dependencies* among index points of data fields are introduced by referencing the values of some data fields on some index points. These index domains eventually have to be distributed among processors of a target parallel machine. The data dependencies account for *communication* among processors if the involved index points are distributed to different processors. In order to minimize the communication cost, the compiler has to find a way to align (if the involved domains are of different size and dimension) and map the index domains to the processors in a way that the resulted computation is efficient.

The description above may be very vague. Let us jump ahead and take the Crystal program `mm3.cr` in Figure 6 as an example. There are six data fields in `mm3.cr`: $A, B, C, a, b,$ and c. The first three are defined over index domain D^2, and the last three are over index domain $D^2 \times \mathsf{interval}(0, n)$. Data fields assign each index point in its domain a value, which is defined by the body of the function. The value of $c(i, j, k)$, as shown in its definitions, *depends* on the values $a(i, j, k), b(i, j, k)$, and $c(i, j, k-1)$. When implementing this program on a parallel machine, we need to map index domains D^2 and $D^2 \times \mathsf{interval}(0, n)$ to processors in sunch a way that the resulting computation is efficient.

2.3 Manipulating Crystal Structures in Meta-Crystal

In order to manipulate Crystal programs, we need to treat Crystal programs, definitions, and expressions as first class objects in Meta-Crystal. We would like to clarify two basic but often confusing concepts pertaining to meta level processing. We call a syntactically correct sequence of characters that represents a program, a definition, or an expression a *notation*. For example, the expressions "$\lambda x.x$" and "3" are notations. There are *structures* corresponding to notations. Structures can be thought of as abstract syntax trees for notations. For clarity of presentation, we use the slanted font for structures. For example, the structure of notation "$(\lambda x.x)(3)$" is *(λx.x+1)(3)*. Structures enjoy some algebraic properties and have a set of constructors, selectors, predicates, and operators associated with them, while notations are just strings of characters. For example, we have a predicate to test whether *(λx.x+1)(3)* is an application. From *(λx.x+1)(3)*, we can select its rator (i.e., the function, or equivalently, the operator), which is *λx.x+1*; and its rand (i.e., operands), which is *3*. We also have an operator $\mathsf{normalize}$ for reducing β-redexes and transforming *(λx.x+1)(3)* to *3+1*.

Let φ be any Crystal notation, we use "'φ'" as the Meta-Crystal notation to denote the Crystal structure for φ. Only Crystal programs, definitions, and expressions can be quoted by '_'. Let τ be any meta expression, the meaning of "$[\![\tau]\!]$" inside a quoted Crystal notation is the Crystal notation of the Crystal structure to which τ is *evaluated*. That is, '$\ldots [\![\tau]\!] \ldots$' denotes '$\ldots \psi \ldots$', where ψ is the Crystal notation of the value of τ. The evaluation of τ dereferences the meta variables in τ. The scoping is lexical. We call $[\![\_]\!]$ *unquoting*.

We will use Greek letters such as $\alpha, \beta, \kappa, \tau$ and ϕ for meta variables having Crystal structures as their values. Meta-Crystal operators on Crystal structures will be written in sans-serif font for clarity. For example, given the following meta definitions:

$$\alpha = \text{`}(\lambda x.x)(3)\text{'}, \ \beta = \mathsf{normalize}(\alpha), \ f = \lambda x.\text{`}[\![x]\!] + [\![x]\!]\text{'}.$$

The meta variable α is bound to the structure *(λx.x)(3)*, β is bound to the normalized structure *3*, and f is bound to a meta function. We have

$$f(\text{`}1\text{'}) = \text{`}1+1\text{'}, \ f(\alpha) = \text{`}(\lambda x.x)(3) + (\lambda x.x)(3)\text{'}, \ f(\beta) = \text{`}3+3\text{'}.$$

Notice that an unquoted meta expression is only meaningful inside a quoted Crystal expression and it must evaluate to a Crystal structure. Arbitrary nesting of quoting and unquoting is allowed in Meta-Crystal. Meta variables in nested unquoted meta expressions are lexically scoped by its enclosing meta context, regardless of the quoted Crystal expression. For example, given the meta definitions below:

$$\alpha = \text{`}1\text{'}, \ f = \lambda x.\text{`}(\lambda x.[\![x]\!] + [\![\alpha]\!])(A)\text{'}.$$

The unquoted x in the definition of f is bound to the leftmost occurrence of x, the formal argument of the meta function. We have

$$f(\text{`}x\text{'}) = \text{`}(\lambda x.x+1)A\text{'},\ \ \mathsf{normalize}(f(\text{`}x\text{'})) = \text{`}A+1\text{'},$$
$$f(\text{`}y\text{'}) = \text{`}(\lambda x.y+1)A\text{'},\ \ \mathsf{normalize}(f(\text{`}y\text{'})) = \text{`}y+1\text{'}.$$

A set of operators for transforming Crystal structures algebraically is provided in Meta-Crystal. For example, normalize(κ) β-reduces all β-redexes in κ; and subst(κ, τ_1, τ_2) substitutes all *free* occurrences of τ_1 in κ with τ_2. For example, let $\circ$ denote the function composition and given two meta definitions $\kappa = \text{`}f\circ((\lambda f.f(x))(g))\text{'}$ and $\phi = \text{`}h\text{'}$, we have subst($\kappa$, '$f$', ϕ) = '$h\circ((\lambda f.f(x))(g))$', and normalize($\kappa$) = '$f\circ(g(x))$'. Let $e_1, e_2, \ldots$ range over Meta-Crystal expressions, κ over Crystal definitions, f over Crystal function names, and ρ over Crystal programs, some of the operators used in the examples in next section are defined below:

parse-file(`file`) Denotes the definitions and the output expression in the Crystal program `file`.

produce-file(`file`, $e_1, e_2, \ldots$) Writes the notations denoted by $e_1, e_2, \ldots$ to the Crystal program `file`.

output-exp(ρ) Picks out the output expression from the program ρ.

def(f, ρ) Denotes the definition, which is contained in ρ, that defines f.

expand(κ, f, ρ) Substitutes all free occurrence of f in κ with the right hand side of the definition of f in ρ.

unfold(κ, f, ρ) Does normalize(expand(κ, f, ρ)).

simplify-arith(κ) Simplifies the arithmetic and boolean expressions into some canonical form.

3 Derivation of A Class of Matrix Multiplication Programs

In this section we use matrix multiplication as an example to demonstrate how a class of different programs can be derived. Assuming the compiler generates code for a distributed-memory message-passing parallel machine based on some popular interconnection network (e.g., hypercube or mesh) and the size of the matrices is so large that replicating them on every processor is undesirable, these programs pose various degrees of difficulty for the compiler to produce efficient code. Starting from a simple, intuitive, though inefficient program, a sophisticated and efficient program can be derived by program transformation using Meta-Crystal.

3.1 Initial Program

Consider the Crystal program `mm1.cr` in Figure 1 which computes the matrix product, $C = A \times B$, where A and B are matrices of size n by n and are read in from the input. D is an interval index domain from 1 to

$$\begin{array}{rcl}
A &=& \lambda(i,j): D^2.\text{read matrix},\ \ B = \lambda(i,j): D^2.\text{read matrix},\\
D &=& \mathsf{interval}(1,n),\\
C &=& \lambda(i,j): D^2.\mathsf{reduce}(+,\ 0,\ (\lambda k: D.A(i,k) * B(k,j)))\\
? & C &
\end{array}$$

Figure 1: Initial matrix multiplication program `mm1.cr`.

n. A and B are data fields over D^2, which is the usual cartesian product $D \times D$. C is a data field over D^2, indexed by (i, j), with a reduction expression as its body. The primitive function **reduce** is a higher-order function whose general form of its application is "$\mathsf{reduce}(\oplus, \mathrm{Id}_\oplus, \text{data-field})$". Its first argument, $\oplus$, is a binary, associative operator, which is addition in this example. Its second argument, $\mathrm{Id}_\oplus$, is the identity element of $\oplus$, which is 0 in this example. Its third argument, "data-field", is a data field. Let F be a data field containing a set of values $x_1, x_2, \ldots, x_n$, then "$\mathsf{reduce}(\oplus, \mathrm{Id}_\oplus, F)$" is defined to be $x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

Program `mm1.cr` is just a straightforward translation of the definition for the matrix product: $C(i,j) = \sum_{k \in D}(A(i,k) \times B(k,j))$. The matrices A, B and C are three two-dimensional data fields which are to be aligned and then distributed among the processors by the compiler. Without replicating A and B, no matter how these data fields are aligned and distributed there will be many long distance data movements required for computing the values of C, as shown in Figure 2. To generate a good parallel implementation for this program, the compiler needs to know how to implement reduction smartly on the target machine. In general, this is a rather complicated task on distributed memory parallel machines because the long distance data movements involved make the task of minimizing communication cost difficult.

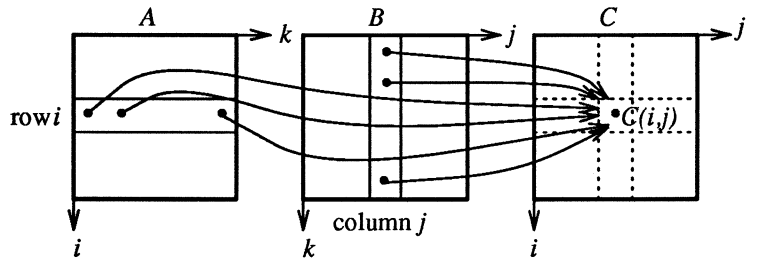

Figure 2: Data movements needed for computing reduction at $C(i, j)$.

3.2 Reduction Elimination

We can help the compiler by getting rid of the hard-to-compile reduction construct. Furthermore, because the neighborhood communication is usually much less expensive on hypercube or mesh than long distance communication, trying to make the data movements take place only among neighbors of each indexed point will be beneficial as well. There are systematic ways to transform reduction expressions. Consider a general reduction expression below:

$$\mathsf{reduce}(\oplus, \mathrm{Id}_\oplus, (\lambda k : D.\tau[k]))$$

where $\oplus$ and $\mathrm{Id}_\oplus$ are as described above, $D = \mathsf{interval}(l, u)$, and $\tau[k]$ is some expression with occurrences of the formal argument k. Then, $\lambda k : D.\tau[k]$ is the data field over which reduction is performed. One way to eliminate the reduction is by introducing a temporary data field over the domain "$\mathsf{interval}((l-1), n)$" to compute the reduction linearly. Simple induction over $\mathsf{interval}((l-1), n)$ shows that the expression below yields the same value as the reduction expression above:

$$f(h) \text{ where}\{\ f = \lambda k : \mathsf{interval}((l-1), n). \begin{Bmatrix} k = l - 1 \rightarrow \mathrm{Id}_\oplus \\ l \le k \le h \rightarrow f(k-1) \oplus \tau[k] \end{Bmatrix} \},$$

where f is the name for the temporary data field and the notation "$k = l - 1$" in the conditional is the equality testing between k and $l - 1$. A meta procedure $\mathsf{elim\text{-}reduction}(\kappa, f)$ can be written in Meta-Crystal to look into the Crystal structure κ recursively for an application structure with **reduce** as its rator, and then transform the application structure as described above by using f as the name of the temporary data field. Due to the space limitation the actual Meta-Crystal code for **elim-reduction** will not be presented here and it will be used as a built-in operator. The derivation to transform the reduction expression follows:

1. Read in the program `mm1.cr`: $\rho =$ parse-file(`mm1.cr`).
2. Pick out the definition of C from ρ:
   $\kappa_1 = \text{def}(\text{`}C\text{'}, \rho) = \text{`}C = \lambda(i,j) : D^2.\text{reduce}(+, 0, (\lambda k : D.A(i,k) * B(k,j)))\text{'}$
3. Eliminate the reduction and simplify the arithmetics:

$$\begin{aligned}\kappa_2 &= \text{simplify-arith}(\text{elim-reduction}(\kappa_1, \text{`}c\text{'})) \\ &= \text{`}C = \lambda(i,j) : D^2.c(n) \\ &\qquad \text{where}\{\ c = \lambda k : \text{interval}(0,n).\begin{Bmatrix} k = 0 \to 0 \\ 1 \le k \le n \to c(k-1) + A(i,k) * B(k,j) \end{Bmatrix}\}\text{'}\end{aligned}$$

4. The local definition of c in the definition of C contains free variables i and j that are part of the formal arguments of C. Preparing for the next step, we would like to make free variables i and j locally bound in c. This is done by the operator expand-dom-of-local-def(κ, 'C', 'c'). It looks in κ for local definition c inside C. If c has any free variables which are part of C's formal arguments, it expands the domain of c to include the domains over which those free variables range and transforms all the applications of c by augmenting the new arguments. That is, we can explicitly pass those free variables through the new arguments of c and the meaning of C stays intact. The derivation below shows how this operator works:

$$\begin{aligned}\kappa_3 &= \text{expand-dom-of-local-def}(\kappa_2, \text{`}C\text{'}, \text{`}c\text{'}) \\ &= \text{`}C = \lambda(i,j) : D^2.c(i,j,n) \\ &\qquad \text{where}\{\ c = \lambda(i,j,k) : (D^2 \times \text{interval}(0,n)).\begin{Bmatrix} k = 0 \to 0 \\ 1 \le k \le n \to c(i,j,k-1) + A(i,k) * B(k,j) \end{Bmatrix}\}\text{'}\end{aligned}$$

5. Make c a global definition. This is done by the operator flatten-local-def(κ, 'C', 'c'). It looks in κ for local definition c inside C. If (1) c does not contain any free variable which is one of C's formal arguments; (2) c does not contain any function application using other local function of C; (3) c does not introduce name conflict with other global definitions, then c is moved out of C.

$$\begin{aligned}\kappa_4 &= \text{flatten-local-def}(\kappa_3, \text{`}C\text{'}, \text{`}c\text{'}) \\ &= \text{`}C = \lambda(i,j) : D^2.c(i,j,n), \\ &\qquad c = \lambda(i,j,k) : (D^2 \times \text{interval}(0,n)).\begin{Bmatrix} k = 0 \to 0 \\ 1 \le k \le n \to c(i,j,k-1) + A(i,k) * B(k,j) \end{Bmatrix}\text{'}\end{aligned}$$

6. Produce program `mm2.cr`:
   produce-file(mm2.cr, def('D', ρ), def('A', ρ), def('B', ρ), κ_4, output-exp(ρ)).

Figure 3 shows the Crystal program `mm2.cr` produced by the transformations. The Meta-Crystal program for producing `mm2.cr` is the concatenation of all the meta definitions introduced in steps 1 to 5, together with the output expression in step 6. Figure 4 shows one possible parallel interpretation of this program. The new data field c is a three-dimensional data field indexed by (i,j,k). We conveniently place the data field C on the i-j plane where $k = n$, the data field A on the i-k plane where $j = 1$, and the data field B on the j-k plane where $i = 1$. The data movements are somewhat more regular than the first program, but long distance communications are still required because each $A(i,k)$ are referenced by all $j, 1 \le j \le n$, and each $B(k,j)$ are referenced by all $i, 1 \le i \le n$. That is, values of $A(i,k)$ need to be *broadcast* along the j dimension, and similarly for $B(k,j)$ along the i dimension. These communications are still expensive on hypercube or mesh and it is not easy for the compiler to minimize the communication cost.

$$\begin{aligned}
D &= \mathsf{interval}(1,n),\quad A = \lambda(i,j): D^2.\text{read matrix},\quad B = \lambda(i,j): D^2.\text{read matrix},\\
C &= \lambda(i,j): D^2.c(i,j,n),\\
c &= \lambda(i,j,k): D^2 \times \mathsf{interval}(0,n).\left\{\begin{array}{l} k=0 \rightarrow 0 \\ 1 \le k \le n \rightarrow c(i,j,k-1) + A(i,k) * B(k,j) \end{array}\right\},\\
?\ C
\end{aligned}$$

Figure 3: Program `mm2.cr` with reduction removed.

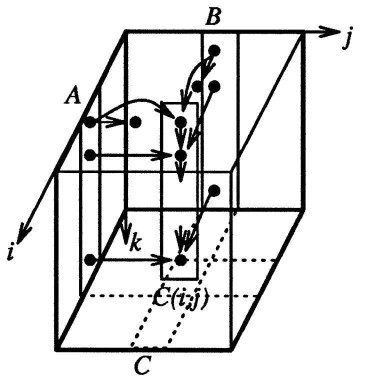

Figure 4: Parallel Interpretation of `mm2.cr`. Values of $A(i,k)$ and $B(k,j)$ are broadcast along the j and i dimension, respectively.

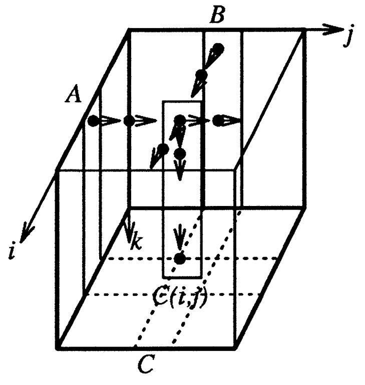

Figure 5: Parallel Interpretation of `mm3.cr`. Values of $A(i,k)$ and $B(k,j)$ are propagated by $a(i,j,k)$ and $b(i,j,k)$, respectively.

3.3 Broadcast Elimination

One way to reduce the long distance communications in `mm2.cr` is to propagate the values of A along the j dimension (starting from $j = 1$) and B along the i dimension (starting from $i = 1$) with new data fields a and b defined over $D^2 \times \mathsf{interval}(0,n)$. This can be done by the **elim-broadcast** operator. Elim-broadcast$(\kappa, f, g, \hat{g})$ looks in κ for any usage of the data field g in the definition for the data field f. Suppose the domain of g is D_g and the domain of f is D_f, if D_g is one dimension lower than D_f, then a new data field $\hat{g}$ over D_f is created to propagate the valued of g along the missing dimension, starting from the lower bound of that dimension. Simple induction over the missing dimension shows that the meaning of f stays intact. Derivations below give examples of how elim-broadcast works.

1. Read in `mm2.cr`: $\rho = \mathsf{parse\text{-}file}(\texttt{mm2.cr})$.
2. Pick out the definition of c:

$$\begin{aligned}
\kappa_1 &= \mathsf{def}(`c\text{'},\rho)\\
&= `c = \lambda(i,j,k): (D^2 \times \mathsf{interval}(0,n)).\left\{\begin{array}{l} k=0 \rightarrow 0 \\ 1 \le k \le n \rightarrow c(i,j,k-1) + A(i,k) * B(k,j) \end{array}\right\}\text{'},
\end{aligned}$$

3. Eliminate broadcasting of $A(i,k)$ and $B(k,j)$ in c by creating a and b:

$$\begin{aligned}
\kappa_2 &= \mathsf{elim\text{-}broadcast}(\mathsf{elim\text{-}broadcast}(\kappa_1, `c\text{'}, `A\text{'}, `a\text{'}), `c\text{'}, `B\text{'}, `b\text{'})\\
&= `c = \lambda(i,j,k): (D^2 \times \mathsf{interval}(0,n)).\left\{\begin{array}{l} k=0 \rightarrow 0 \\ 1 \le k \le n \rightarrow c(i,j,k-1) + a(i,j,k) * b(i,j,k) \end{array}\right\}
\end{aligned}$$

$$\text{where}\{a = \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} j = 1 \rightarrow A(i,k) \\ 1 < j \leq n \rightarrow a(i,j-1,k) \end{Bmatrix},$$

$$b = \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} i = 1 \rightarrow B(k,j) \\ 1 < i \leq n \rightarrow b(i-1,j,k) \end{Bmatrix}\}\text{'}$$

4. Flatten local definitions a and b in c:

$$\begin{aligned} \kappa_3 &= \mathsf{flatten\text{-}local\text{-}def}(\mathsf{flatten\text{-}local\text{-}def}(\kappa_2, \text{`}c\text{'}, \text{`}a\text{'}), \text{`}c\text{'}, \text{`}b\text{'}) \\ &= \text{`}a = \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} j = 1 \rightarrow A(i,k) \\ 0 < j \leq n \rightarrow a(i,j-1,k) \end{Bmatrix}, \\ &\quad b = \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} i = 1 \rightarrow B(k,j) \\ 0 < i \leq n \rightarrow b(i-1,j,k) \end{Bmatrix}, \\ &\quad c = \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} k = 0 \rightarrow 0 \\ 0 < k \leq n \rightarrow a(i,j,k) \times b(i,j,k) + c(i,j,k-1) \end{Bmatrix}\text{'} \end{aligned}$$

5. Produce program mm3.cr:
$\mathsf{produce\text{-}file}(\mathtt{mm3.cr}, \mathsf{def}(\text{`}D\text{'}, \rho), \mathsf{def}(\text{`}A\text{'}, \rho), \mathsf{def}(\text{`}B\text{'}, \rho), \mathsf{def}(\text{`}C\text{'}, \rho), \kappa_3, \mathsf{output\text{-}exp}(\rho))$.

Figure 6 shows program mm3.cr produced by the derivation above. Figure 5 shows one straightforward parallel interpretation of this program. We place data field a, b, and c in the same three-dimensional domain indexed by (i,j,k), and place A, B, and C as before. Now all the data movements are local because for any index point (i,j,k), only the values on points $(i-1,j,k)$, $(i,j-1,k)$, and $(i,j,k-1)$ are needed. This program is much easier for the compiler to produce efficient code because, firstly, mapping a three-dimensional mesh onto a hypercube or onto a two-dimensional mesh is rather straightforward, and, secondly, the communication now are all localized and there is no need to minimize long distance communications any more.

$$\begin{aligned} D &= \mathsf{interval}(1,n), \quad A = \lambda(i,j) : D^2.\text{read matrix}, \quad B = \lambda(i,j) : D^2.\text{read matrix}, \\ C &= \lambda(i,j) : D^2.c(i,j,n), \\ a &= \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} j = 1 \rightarrow A(i,k) \\ 0 < j \leq n \rightarrow a(i,j-1,k) \end{Bmatrix}, \\ b &= \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} i = 1 \rightarrow B(k,j) \\ 0 < i \leq n \rightarrow b(i-1,j,k) \end{Bmatrix}, \\ c &= \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).\begin{Bmatrix} k = 0 \rightarrow 0 \\ 0 < k \leq n \rightarrow a(i,j,k) \times b(i,j,k) + c(i,j,k-1) \end{Bmatrix} \\ &?\ C \end{aligned}$$

Figure 6: Program mm3.cr with broadcasting removed.

3.4 Space-Time Transformation with Reshape Morphism

A non-sophisticated compilation of the program `mm3.cr` needs a space of size $O(n^3)$ for the domain $D^2 \times \mathsf{interval}(0,n)$. Examining the data dependencies among the index points, we observe that the computation can proceed in a wavefront fashion as shown in Figure 7. However, all the index points are only used once through the course of the computation. Therefore, the space utilization is sparse. If the space usage is a major concern, we can improve the space utilization by space-time transformation. Consider the following linear mappings:

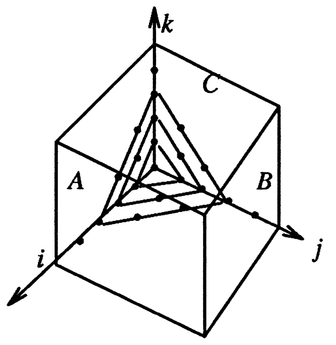

Figure 7: Wavefronts of the computation in program `mm3.cr`.

$$\begin{tikzcd}
D_1 \arrow[r, "f"] \arrow[d, shift right, "g"'] & V \\
D_2 \arrow[u, shift right, "g^{-1}"'] \arrow[ur, "\hat{f}"'] &
\end{tikzcd}$$

Figure 8: Commuting diagram of reshape morphism.

$$\begin{aligned}
E &= \mathsf{interval}(1,2n), \quad T = \mathsf{interval}(2,3n) \\
g &= \lambda(i,j,k) : (D^2 \times \mathsf{interval}(0,n)).(i+k, j+k, i+j+k) : (E^2 \times T) \\
g^{-1} &= \lambda(x,y,t) : (E^2 \times T).(t-y, t-x, x+y-t) : (D^2 \times \mathsf{interval}(0,n))
\end{aligned}$$

The function g maps index points in $(D^2 \times \mathsf{interval}(0,n))$ into the product domain of a two-dimensional space E^2 times an explicit time domain T, and g^{-1} does the inverse. Coming up with the mapping g requires some insight: the time component $(i+j+k)$ defines the plane equations of the wavefronts, and the space components $(i+k, j+k)$ spreads out points in $D^2 \times \mathsf{interval}(0,n)$ onto E^2 in a proper way. Given g, E and T can be derived by mapping the extreme points of D through g. The mapping g^{-1} can be derived by solving for i,j,k in the simultaneous linear equation system: $x = i+k, y = j+k, t = i+j+k$.

The task of deriving a new program given such mappings can be generalized as follows. Let D_1, D_2 be two index domains, and $f : D_1 \to V$, $\hat{f} : D_2 \to V$ be two data fields, where V is some value domain. The mapping $g : D_1 \to D_2$ is a *reshape morphism* if it has an inverse $g^{-1} : D_2 \to D_1$ which makes the diagram in Figure 8 commute. Our goal is to derive $\hat{f}$ given f, g, and g^{-1} when computing $\hat{f}$ is more desirable than computing f.

Given these mappings, deriving a new program by hand is rather tedious and error-prone. Meta-Crystal can automate the transformation. Users can work at a higher conceptual level and be sure that the derived program is correct with respect to the input program and mappings between index domains. A meta-procedure for deriving $\hat{f}$ by utilizing the equality $\hat{f} = f \circ g^{-1}$, unfolding of g and g^{-1}, and some equational transformations is developed in [6]. A simplified version is presented in the following derivations for $\hat{a}$, where $\hat{a} = a \circ g^{-1}$. We assume that the definitions above for E, T, g and g^{-1} have been incorporated into `mm3.cr`.

1. Read in program `mm3.cr`: $\rho = \mathsf{parse\text{-}file}(\texttt{mm3.cr})$.

2. Do η-abstraction[1] on $a \circ g^{-1}$ in the equation '$\hat{a} = a \circ g^{-1}$', which is valid by definition: $\kappa_1 =$ '$\hat{a} = \lambda(x,y,t) : (E^2 \times T).(a \circ g^{-1}(x,y,t))$'
3. Unfold the composition: $\kappa_2 = \mathsf{unfold}(\kappa_1, \text{'} \circ \text{'}, \rho) =$ '$\hat{a} = \lambda(x,y,t) : (E^2 \times T).a(g^{-1}(x,y,t))$'
4. Unfold g^{-1}: $\kappa_3 = \mathsf{unfold}(\kappa_2, \text{'}g^{-1}\text{'}, \rho) =$ '$\hat{a} = \lambda(x,y,t) : (E^2 \times T).a(t-y, t-x, x+y-t)$'
5. Unfold a:

$$\begin{aligned} \kappa_4 &= \mathsf{unfold}(\kappa_3, \text{'}a\text{'}, \rho) \\ &= \text{'}\hat{a} = \lambda(x,y,t) : (E^2 \times T).\begin{Bmatrix} t-x=1 \rightarrow A(t-y, x+y-t) \\ 0 < t-x \le n \rightarrow a(t-y, t-x-1, x+y-t) \end{Bmatrix}\text{'} \end{aligned}$$

6. Remove a in the definition of $\hat{a}$ by substituting a with $\hat{a} \circ g$:

$$\begin{aligned} \kappa_5 &= \mathsf{subst}(\kappa_4, \text{'}a\text{'}, \text{'}\hat{a} \circ g\text{'}) \\ &= \text{'}\hat{a} = \lambda(x,y,t) : (E^2 \times T).\begin{Bmatrix} t-x=1 \rightarrow A(t-y, x+y-t) \\ 0 < t-x \le n \rightarrow (\hat{a} \circ g)(t-y, t-x-1, x+y-t) \end{Bmatrix}\text{'} \end{aligned}$$

7. Unfold the composition, then unfold g, then simplify the arithmetics:

$$\begin{aligned} \kappa_6 &= \mathsf{simplify\text{-}arith}(\mathsf{unfold}(\mathsf{unfold}(\kappa_5, \text{'} \circ \text{'}, \rho), \text{'}g\text{'}, \rho)) \\ &= \text{'}\hat{a} = \lambda(x,y,t) : (E^2 \times T).\begin{Bmatrix} t=x+1 \rightarrow A(t-y, x+y-t) \\ 0 < t-x \le n \rightarrow \hat{a}(x, y-1, t-1) \end{Bmatrix}\text{'} \end{aligned}$$

Similarly, we can derive $\hat{b}$ and $\hat{c}$, where $\hat{b} = b \circ g^{-1}$ and $\hat{c} = c \circ g^{-1}$. We can abstract over 'a', '$\hat{a}$', 'g' and 'g^{-1}', and generalize this derivation into a meta procedure **reshape** as shown in Figure 10. Using **reshape**, $\hat{a}$ can be derived by reshape('a', '$\hat{a}$', 'g', 'g^{-1}', ρ); $\hat{b}$ can be derived by reshape('b', '$\hat{c}$', 'g', 'g^{-1}', ρ); and $\hat{c}$ can be derived by reshape('c', '$\hat{c}$', 'g', 'g^{-1}', ρ), followed by a few steps to replace occurrences of a and b with $\hat{a}$ and $\hat{b}$: first substitute a and b with $\hat{a} \circ g$ and $\hat{b} \circ g$, respectively, then unfold the composition, finally unfold g. Similarly, for replacing the occurrence of c in C with $\hat{c}$. The derived program `mm4.cr` is shown in Figure 9. It needs a space of size $4n^2$ for the two-dimensional space E^2, indexed by (x,y), as opposed to the size of $O(n^3)$ in program `mm3.cr`. The parallel interpretation of `mm4.cr` is a sophisticated parallel matrix multiplication program. From the definition of C, we can see that the result is scattered all over the domain $E^2 \times T$. Index points of E^2 are re-used through successive steps of computation for $\hat{c}$, as opposed to being only used once in `mm3.cr`. This program is also efficient because all the dependencies are local. Assuming the definitions of E, T, g, and g^{-1} have been added into `mm3.cr`, a meta program that transforms `mm3.cr` into `mm4.cr` is shown in Figure 10.

4 Concluding Remarks

The philosophy motivating the work described in this paper can be summarized as follows: we feel that, firstly, program transformation is an effective methodology for deriving more efficient programs (in the sense that the compiler can handle them better) and, secondly, no parallelizing compiler is able to perform well for all input programs. Therefore, a metalanguage which allows the user to synthesize their transformation

[1]Doing *η-abstraction* on any f results in an equivalent function $\lambda x.(fx)$. That is, f is equivalent to $\lambda x.(fx)$ by η-abstraction.

$$\begin{aligned}
D &= \mathsf{interval}(1,n),\ \ A = \lambda(i,j) : D^2.\text{read matrix},\ \ B = \lambda(i,j) : D^2.\text{read matrix},\\
C &= \lambda(i,j) : D^2.\hat{c}(i+n, j+n, i+j+n),\ \ E = \mathsf{interval}(1,2n),\ \ T = \mathsf{interval}(2,3n),\\
\hat{a} &= \lambda(x,y,t) : E^2 \times T.\left\{\begin{array}{l} t = x+1 \rightarrow A(t-y, x+y-t) \\ 0 < t-x \leq n \rightarrow \hat{a}(x, y-1, t-1) \end{array}\right\},\\
\hat{b} &= \lambda(x,y,t) : E^2 \times T.\left\{\begin{array}{l} t = y+1 \rightarrow B(t-y, x+y-t) \\ 0 < t-y \leq n \rightarrow \hat{b}(x-1, y, t-1) \end{array}\right\},\\
\hat{c} &= \lambda(x,y,t) : E^2 \times T.\left\{\begin{array}{l} x+y = t \rightarrow 0 \\ 0 < x+y-t \leq n \rightarrow \hat{c}(x-1, y-1, t-1) + \hat{a}(x,y,t) \times \hat{b}(x,y,t) \end{array}\right\},\\
?\ & C
\end{aligned}$$

Figure 9: Program `mm4.cr` after space-time transformation.

$$\begin{aligned}
\mathsf{reshape} &= \lambda(\alpha, \hat{\alpha}, \phi, \phi^{-1}, \rho).\ \ \gamma \ \ \mathsf{where}\{\\
&\qquad \kappa_1 = \text{‘}[\![\hat{\alpha}]\!] = \lambda(x,y,t) : (E^2 \times T).(([\![\alpha]\!] \circ [\![\phi^{-1}]\!])(x,y,t))\text{’},\\
&\qquad \kappa_2 = \mathsf{unfold}(\kappa_1, \text{‘} \circ \text{’}, \rho),\ \ \kappa_3 = \mathsf{unfold}(\kappa_2, \phi^{-1}, \rho),\ \ \kappa_4 = \mathsf{unfold}(\kappa_3, \alpha, \rho),\\
&\qquad \kappa_5 = \mathsf{subst}(\kappa_4, \alpha, \text{‘}[\![\hat{\alpha}]\!] \circ [\![\phi]\!]\text{’}),\ \ \gamma = \mathsf{simplify\text{-}arith}(\mathsf{unfold}(\mathsf{unfold}(\kappa_5, \text{‘} \circ \text{’}, \rho), \phi, \rho))\ \},\\
\rho &= \mathsf{parse\text{-}file}(\mathtt{mm3.cr}),\\
\kappa_{\hat{a}} &= \mathsf{reshape}(\text{‘}a\text{’}, \text{‘}\hat{a}\text{’}, \text{‘}g\text{’}, \text{‘}g^{-1}\text{’}, \rho),\ \ \kappa_{\hat{b}} = \mathsf{reshape}(\text{‘}b\text{’}, \text{‘}\hat{b}\text{’}, \text{‘}g\text{’}, \text{‘}g^{-1}\text{’}, \rho),\\
\kappa_1 &= \mathsf{reshape}(\text{‘}c\text{’}, \text{‘}\hat{c}\text{’}, \text{‘}g\text{’}, \text{‘}g^{-1}\text{’}, \rho),\ \ \kappa_2 = \mathsf{subst}(\mathsf{subst}(\kappa_1, \text{‘}a\text{’}, \text{‘}\hat{a} \circ g\text{’}), \text{‘}b\text{’}, \text{‘}\hat{b} \circ g\text{’}),\\
\kappa_{\hat{c}} &= \mathsf{unfold}(\mathsf{unfold}(\kappa_2, \text{‘} \circ \text{’}, \rho), \text{‘}g\text{’}, \rho),\\
\kappa_3 &= \mathsf{subst}(\kappa_1, \text{‘}c\text{’}, \text{‘}\hat{c} \circ g\text{’}),\ \ \kappa_C = \mathsf{unfold}(\mathsf{unfold}(\kappa_3, \text{‘} \circ \text{’}, \rho), \text{‘}g\text{’}, \rho),\\
? \quad & \mathsf{produce\text{-}file}(\mathtt{mm4.cr}, \mathsf{def}(\text{‘}D\text{’}, \rho), \mathsf{def}(\text{‘}A\text{’}, \rho), \mathsf{def}(\text{‘}B\text{’}, \rho), \kappa_C, \mathsf{def}(\text{‘}E\text{’}, \rho), \mathsf{def}(\text{‘}T\text{’}, \rho),\\
&\qquad \kappa_{\hat{a}}, \kappa_{\hat{b}}, \kappa_{\hat{c}}, \mathsf{output\text{-}exp}(\rho))
\end{aligned}$$

Figure 10: Meta program that transforms `mm3.cr` into `mm4.cr`.

schemes at a high conceptual level and assists in transforming programs should be very useful. Meta-Crystal is designed to provide such a tool. As a general programming language, it is relatively flexible, extensible, expressive, and powerful compared to other program transformation systems. It is also clean and simple.

Some readers may feel that all of the transformations presented in this paper can be done at the optimization phase of a very smart compiler. It is correct to the extent that we have to realize all these optimization tricks and design the compiler to apply them in the right occasion, all *before* the compiler is released. Having a metalanguage like Meta-Crystal allows us to implement transformation tricks that we discover later and to deal with cases that may be too special to incorporate in a reasonably efficient compiler. Sophisticated programmers may find it useful for building their own library of optimization schemes as well.

Meta-Crystal has been implemented in T (a dialect of Scheme) and has been actually in use. An interactive environment similar to a Lisp interpreter has been built. It has proved to be a very useful tool for quickly deriving new programs for other examples. Users can concentrate on the creative phase of the transformation and are relieved from the mechanical and error-prone algebraic manipulations. Many ideas used in the design of this metalanguage (such as quoting, unquoting, various semantic-preserving operators on structures, normalization, and η-abstraction) are applicable to other languages for building their own program manipulation and transformation systems. We believe that such a metalanguage and an interactive environment is a useful tool for employing program transformation as a methodology for developing efficient parallel programs.

References

[1] F. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations – computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, Feb. 1989.

[2] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[3] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[4] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[5] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2(2):171–207, October 1988.

[6] Young-il Choo and Marina Chen. A theory of parallel-program optimization. Technical Report YALEU/DCS/TR-608, Dept. of Computer Science, Yale University, July 1988.

[7] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941.

[8] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[9] John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[10] Martin S. Feather. A system for assisting program transformation. *ACM Transaction on Programming Language and Systems*, pages 1–20, January 1982.

[11] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the Association for Computing Machinery*, 24(1):121–145, January 1977.

[12] R. Milner. A proposal for standard ML. In *ACM Symp. on LISP and Functional Programming*, pages 184–197, Aug. 1984.

[13] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[14] J. Allan Yang and Young-il Choo. Meta-Crystal – a metalanguage for parallel-program optimization. Technical Report YALEU/DCS/TR-786, Dept. of Computer Science, Yale University, April 1990.