Metalinguistic Features for Formal Parallel-Program Transformation

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

Abstract

This paper describes a set of metalinguistic features and their applications in formal parallel-program transformation. Program transformation is an effective methodology for developing correct and efficient parallel programs. However, manually carrying out the transformation is usually very cumbersome and error-prone. These metalinguistic features provide convenient language constructs for expressing and automating transformation steps. These features include a rationalized version of quoting and unquoting, a set of constructors, selectors, predicates and a collection of semantics preserving operators. Their applications are demonstrated in the formal derivation of a class of parallel programs for solving the palindrome recognition problem.

1 Introduction

As various parallel machines have become available in the past few years, how to effectively program these machines becomes a pressing issue. Crystal [1] is a parallel language aimed at addressing this issue with a high-level functional language and a smart compiler. Sophisticated parallelizing compilation technologies for Crystal have been devised for some commercially available parallel machines [14, 12, 13]. 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 with nonlocal communication into programs with local communication [6]. Though the intuition behind the transformations is not too difficult, the technicalities required to derive the new programs are often quite involved and cumbersome. A metalanguage has been presented to automate the mechanical aspect of the transformations in [7]. As a result, a set of metalinguistic features are designed to provide a formal, expressive, and easy-to-use tool for expressing transformational strategies as meta procedures that can be used repetitively on similar occasions.

This paper describes these metalinguistic features and their applications in expressing formal parallel-program transformation. These features include a rationalized version of quoting and unquoting, a set of constructors, selectors and predicates, and a collection of semantics preserving operators. This paper is organized as follows. Section 2 briefly introduces the abstract syntax of Crystal. Section 3 describes the metalinguistic features, illustrates how Crystal programs are conveniently manipulated with these meta constructs, and presents a formal denotational semantics of the rationalized quoting and unquoting. Section 4 briefly introduces how Crystal programs are interpreted on parallel machines so that readers can understand the examples presented in Section 5. Section 5 demonstrates the applications of these metalinguistic features by formally deriving a class of parallel palindrome recognition programs as an example. Section 6 compares our work with some related works. Section 7 provides some concluding remarks.

2 Abstract Syntax of Crystal

Crystal is a strongly-typed, lexically scoped functional language with a simple syntax. Its syntax is based on λ-calculus [1], enriched with conditional, recursion, and local definitions. Let \( x, \) range over identifiers, \( \epsilon, \) and \( g, \) range over expressions:

- A program is a set of mutually recursive definitions with an output expression:

  \[ x_1=\epsilon_1, \ x_2=\epsilon_2, \ldots, \ ?\epsilon. \]

  The notation \( x_1=\epsilon_1 \) is a definition that binds the identifier \( x_1 \) to the value of \( \epsilon_1. \) The result of the program is the value of the output expression \( ?\epsilon. \) The order of definitions is irrelevant.
Local definitions can be introduced to an expression with the where {...} construct:

\[
e \text{where} \{ \, z_1 \mapsto e_1, \ z_2 \mapsto e_2, \ \ldots \, \} .
\]

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

The conditional expressions have the following form: 
\[
e_1 \text{ if } e \text{ then } e_1 \text{ else } e_2.
\]

Provided that the language of interest is unambiguous, there is a unique structure for every notation. There could be more than one notation corresponding to a particular structure. For example, \( \beta \) could be the structure of 3, \( \langle 3 \rangle \), etc. But these notations should all denote the same structure, and we can pick any of these to be the notation of the structure \( \beta \). Therefore, by "the notation of the structure \( \beta \)" we mean any notation which has \( \beta \) as its structure.

3.2 Rationalized Quoting and Unquoting

Let \( \psi \) be any Crystal notation, we use \( '\psi' \) as the notation to denote the Crystal structure for \( \psi \). We call \( ' \cdot ' \) quoting. Only programs, definitions, and expressions can be quoted. Let \( r \) be any expression denoting a structure, the meaning of \( (r) \) inside a quoted notation is the notation of the structure to which \( r \) is evaluated. That is, \( \langle \cdots (r) \cdots \rangle \) denotes \( \langle \cdots \psi \cdots \rangle \), where \( \psi \) is the notation of the value of \( r \). The evaluation of \( r \) dereferences the variables in \( r \). The scoping is lexical (explained further in the last paragraph of this subsection). We call \( \langle \cdot \rangle \) unquoting.

We use Greek letters such as \( \alpha, \beta, \) and \( \kappa \) for variables that range over structures. Operators on structures will be written in sans serif font for clarity. For example, given the following definitions

\[
\alpha = (\lambda x. z)(3), \quad \beta = \text{normalize}(\alpha), \quad f = \lambda x. (z) + (x)
\]

the variable \( \alpha \) is bound to the structure \( \lambda x. z)(3) \), \( \beta \) is bound to the normalized structure 3, and \( f \) is bound to a function. We have

\[
f(\alpha) = (\lambda x. z)(3) + (\lambda x. z)(3), \quad f(\beta) = '3 + 3'.
\]

Notice that an unquoted expression is only meaningful inside a quoted notation and it must evaluate to a structure. Arbitrary nesting of quoting and unquoting is allowed. Variables in nested unquoted expressions are lexically scoped by its enclosing environment, regardless of the quoted notation. For example, given

\[
\alpha = '1', \quad f = \lambda x. (\lambda x. z) + (\alpha)(A)
\]

the unquoted \( x \) in the definition of \( f \) is bound to the leftmost occurrence of \( x \), the formal argument of the outermost function. We have

\[
f('x') = (\lambda x. z + 1)A, \quad \text{normalize}(f('x')) = 'A + 1',
\]

\[
f('y') = (\lambda x. y + 1)A, \quad \text{normalize}(f('y')) = 'y + 1'.
\]
3.3 Constructors, Selectors, Predicates and Operators for Structures

A set of constructors, selectors, and predicates are provided for each kind of Crystal structure. For example, for the structure of application we have the following:

```plaintext
make-app(r1, r2) = (r1)(r2),
rat(r1) = r1, rand(r1) = r2,
appl(τ) = {
  true if τ = mk-app(r1, r2) for some r1, r2,
  false otherwise.
```

Since structures have algebraic properties, a set of operators for transforming structures algebraically is provided. For example, normalize(K) P-reduces structures (i.e. expressions denoting structures), of the operators used in the program derivations in the Section 5 are defined below. Let e, range over meta expressions (i.e. expressions denoting structures), f range over function names, and K;

```plaintext
appl?(r) = \begin{cases} 
  \text{true} & \text{if } \tau = \text{mk-app}(r1, r2) \text{ for some } r1, r2, \\
  \text{false} & \text{otherwise.} 
\end{cases}
```

In conventional notations, syntactic objects are enclosed between double brackets with its enclosing environment. Thus \(E[e]\) is the value of e in the environment \(\rho\). Let \(e_1, \varphi\) range over \(E\), \(e\) range over constants, and \(x\) range over identifiers. The semantic function \(E\) is given by:

```plaintext
E[e_1] = ρ[e_1]
E[e_2] = K[e_2] where K maps constants to semantic values
E[\'f\'] = S(U[e_1]|\rho) where S and U are defined below
E[λx.e] = λy.E[e]|y/z
E[e_1(e_2)] = E[e_1]|E[e_2]|\rho
E[e\rightarrow e_1, e_2] = if E[e] then E[e_1]|\rho else E[e_2]|\rho
E[e where\{x_1=e_1, x_2=e_2, \ldots\}] = E[e]|\rho
```

The auxiliary function \(S : E \times E \rightarrow E\) takes a notation without unquotings and returns its structure. For example, \(S(x+3) = x+3\). The auxiliary function \(U : E \times Env \rightarrow Env\) defines the semantics of notations with unquotings and it returns notations without unquotings. Basically, it evaluates the unquoted expression and returns the notation of the expression's value at the place where the unquoted expression occurs, and leaves everything else intact. Let \(e, \tau\) range over \(E\), \(e\) range over constants, and \(x\) range over identifiers. The auxiliary function \(U\) is given by:

```plaintext
U[\{e_1\}] = \{e_1\}
U[\{e_2\}] = \{e_2\}
U[\{e\}] = N(E[e]|\rho) where N is defined below
U[λx.e]|\rho = λx.U[e]|\rho
U[e_1(e_2)]|\rho = U[e_2]|\rho(U[e_1]|\rho)
U[e\rightarrow e_1, e_2]|\rho = U[e_2]|\rho(U[e_1]|\rho)
U[e where\{x_1=e_1, x_2=e_2, \ldots\}]|\rho = U[e_2]|\rho where\{x_1=e_1, x_2=e_2, \ldots\}
```

The auxiliary function \(N : S \times E \rightarrow E\) takes a structure and returns its notation in some canonical form. For example, \(N(x+3) = x+3\). We use Quine's quasi quote \(\{\}\) to emphasize that the value of \(U\) is a notation and the concatenation of notations returned by \(U\) means syntactic concatenation.

In conventional notations, syntactic objects are enclosed between double brackets with its enclosing environment. Thus \(E[e]\) is the value of e in the environment \(\rho\). Let \(e_1, \varphi\) range over \(E\), \(e\) range over constants, and \(x\) range over identifiers. The semantic function \(E\) is given by:

```plaintext
E[e_1] = ρ[e_1]
E[e_2] = K[e_2] where K maps constants to semantic values
E[\'f\'] = S(U[e_1]|\rho) where S and U are defined below
E[λx.e] = λy.E[e]|y/z
E[e_1(e_2)] = E[e_1]|E[e_2]|\rho
E[e\rightarrow e_1, e_2] = if E[e] then E[e_1]|\rho else E[e_2]|\rho
E[e where\{x_1=e_1, x_2=e_2, \ldots\}] = E[e]|\rho
```

The auxiliary function \(N : S \times E \rightarrow E\) takes a structure and returns its notation in some canonical form. For example, \(N(x+3) = x+3\). We use Quine's quasi quote \(\{\}\) to emphasize that the value of \(U\) is a notation and the concatenation of notations returned by \(U\) means syntactic concatenation.

Since the syntax for Crystal programs \(x_1=x_1, x_2=x_2, \ldots, ?e\) is equivalent to the expression \(e\ where\{x_1 = x_1, x_2=x_2, \ldots\}\), the meaning of programs are given by \(E\) as below:

```plaintext
E[x_1=x_1, x_2=x_2, \ldots, ?e] = E[e]|\rho_1
```

where \(\rho_1\) is the least fixed point of the recursive equation: \(\rho_1 = \rho_0(E[e_1]|\rho_1)/x_1, (E[e_2]|\rho_1)/x_2, \ldots\)

where \(\rho_0\) is the initial environment which contains the definitions of all the constructors, selectors, predicates, and operators described above. Figure 1 summarizes the syntactic domains, semantic domains, the semantic function, and the auxiliary functions in Crystal.
4 Parallel Interpretation of Crystal Programs

To help readers understand the derivations in Section 5, here we briefly describe how Crystal programs are interpreted on parallel machines. Readers are referred to [5, 14, 12, 13] 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 set of index points. For example, \( D_1 = \text{interval}(1, m) \) is an interval index domain with \( m \) index points, indexed from 1 to \( m \). Similarly for \( D_2 = \text{interval}(1, n) \). More complicated index domains can be constructed by cartesian product, coproduct, and restriction. The cartesian product of \( D_1 \) and \( D_2 \), denoted \( D_1 \times D_2 \), is an index domain with \( m \times n \) index points, indexed from \((1,1)\) to \((m,n)\). The coproduct of \( D_1 \) and \( D_2 \), denoted \( D_1 + D_2 \), has \( m+n \) points, indexed by \( i_1(1), \ldots, i_1(m) \), and \( i_2(1), \ldots, i_2(n) \), where \( i_1 : D_1 \to (D_1 + D_2) \) and \( i_2 : D_2 \to (D_1 + D_2) \) are the injections. The restriction of \( D_1 \) by a filter (i.e., a boolean function over domain \( D_1 \)) \( f \), denoted \( D_1 \mid f \), is the subdomain of \( D_1 \) in which only the index points passing (i.e., satisfying) the filter are included. For example, \( D_2 = D_1 \mid (\lambda i. \text{even}(i)) \) is a subdomain of \( D_1 \) where only points with even index are included. By definition, the formal argument of the filter (in this example, \( i \)) ranges over the domain to be restricted (in this example, \( D_1 \)).

Data fields defined 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 some target parallel machine. The data dependencies lead to 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 such a way that the resulting computation is efficient.

5 Applications in Formal Program Transformation

In this section we demonstrate how these metalinguistic features greatly facilitate the formal derivation of a class of parallel programs that solve the palindrome recognition problem. We picked this problem because its simple nature allows us to concentrate on the nontrivial transformational aspects of the program derivation without worrying about the algorithmic issues of the example itself. The transformation strategies presented here can be used to derive other more complicated parallel programs, such as matrix multiplication [22], dynamic programming [6], or Dirichlet product [4].

We assume that the parallelizing compiler generates code for some distributed-memory message-passing parallel machines based on some popular interconnection network such as the hypercube or the mesh. On these machines, the long distance communication is much more expensive than the neighborhood communication. We also assume that the problem size is so large that replicating the whole input on every processor is not desirable.

Palindrome Recognition Problem Let \( A \) be a sequence of \( n \) numbers and \( A(i) \) denotes the \( i \)-th element. Let \( A[1] = A_1 \ldots A_n \) be the prefix of \( A \) of length \( n \). We call \( A[u] \) a palindrome if \( A_i = A_{n-i+1} \) for all \( 1 \leq i \leq u \). The problem is to compute whether \( A[u] \) is a palindrome, for all \( 1 \leq u \leq n \).

5.1 Initial Program

Given the problem above, we can come up with the first Crystal program p1 as shown in Figure 2. The program is just a direct translation of the problem and its correctness is easy to verify. In p1, \( D \) is an interval index domain from 1 to \( n \); \( A \) is a data field over \( D \), indexed by \( i \), whose value is read in from the input; \( P \) is another data field over \( D \), indexed by \( u \), with a reduction expression as its body. The primitive function reduce is a higher-order function whose general form is "\( \text{reduce}(\oplus, \text{Id}_D, F) \)" where \( \oplus \) is an binary, associative operator (the logical and in this example); \( \text{Id}_D \) is the identity element of \( \oplus \) (logical true in this example); and \( F \) is a data field ("\( \lambda i. \text{interval}(1, u), A(i) = A[u-i+1] \)" in this example). Let \( F \) be a data field containing a set of values \( x_1, x_2, \ldots, x_n \), "\( \text{reduce}(\oplus, \text{Id}_D, F) \)" is defined to be \( x_1 \oplus x_2 \oplus \cdots \oplus x_n \). Therefore, \( P(u), 1 \leq u \leq n \), is the logical and of all the equality testings between \( A(i) \) and \( A(u-i+1), 1 \leq i \leq u \), which is exactly the problem definition.
Figure 2: Initial program p1 for palindrome recognition.

Figure 3: The irregular long distance data movements needed for computing \( P(u) \) in p1.

### 5.2 Transforming Reduction

The data dependencies in the definition of \( P(u) \) are dependent on all the \( A(i), 1 \leq i \leq n \). To compute \( P(u) \), we have to ship to the processor responsible for computing \( P(u) \), as shown in Figure 3. This will require many irregular data movements. One way to localize these irregular communication is to transform the reduction expressions by utilizing the following equation:

\[
\text{reduce}(\otimes, \text{Id}_\otimes, (\lambda i : \text{interval}(l - 1, u). f(i))) = f(u) \quad \text{where}
\begin{align*}
  & f = \lambda i : \text{interval}(l - 1, u). \\
  & \begin{cases}
    i = l - 1 \rightarrow \text{Id}_\otimes \\
    1 \leq i \leq u \rightarrow f(i - 1) \otimes r[i]
  \end{cases}
\end{align*}
\]

In the first part of the equation, the \( \otimes \) and \( \text{Id}_\otimes \) are as described in the previous subsection; \( D = \text{interval}(l, u) \) for some lower bound \( l \) and upper bound \( u \); \( r[i] \) is some expression with occurrences of the formal argument \( i \). Thus \( \lambda i : \text{interval}(l - 1, u). f(i) \) is the data field over which the reduction is performed. The second part of the equation yields the same value by computing the reduction linearly with a temporary data field \( f \) over the slightly extended index domain \( \text{interval}(l - 1, u) \). Simple induction over \( \text{interval}(l - 1, u) \) can prove the validity of the equation.

A meta procedure elim-reduction\( (\kappa, f) \) can be written to look into the structure \( \kappa \) 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 space limitation, some meta operators, e.g., \( \text{expand-dom-of-local-def} \) and \( \text{flatten-local-def} \), are used in the derivations below without being formally defined. Readers are referred to [22] for more details about them. The derivations to transform the reduction expression follow:

1. Read in the program \( p1 \): \( \rho = \text{parse-file}(p1) \).
2. Pick out the definition of \( P \) from \( \rho \): \( \kappa_1 = \text{def}(\ 'P', \rho) \)
3. Eliminate the reduction and simplify the arithmetics:

\[
\begin{align*}
  & \kappa_2 = \text{elim-reduction}(\kappa_1, 'p') \\
  & = \ 'P' = \lambda u : D.p(u) \text{ where} \\
  & \begin{cases}
    i = 0 \rightarrow \text{true} \\
    1 \leq i \leq u \rightarrow p(u, i - 1) \\
    \text{and } A(i) = A(u - i + 1)
  \end{cases}
\end{align*}
\]

4. Expand the domain of the local definition \( p \) to make the free variable \( u \) bound in the new definition of \( p \), then make \( p \) a global definition:

\[
\begin{align*}
  & \kappa_3 = \text{expand-dom-of-local-def}(\kappa_2, 'P', 'p') \\
  & \kappa_4 = \text{flatten-local-def}(\kappa_3, 'P', 'p') \\
  & = \ 'P' = \lambda u : D.p(u, u) \\
  & \begin{cases}
    i = 0 \rightarrow \text{true} \\
    1 \leq i \leq u \rightarrow p(u, i - 1) \\
    \text{and } A(i) = A(u - i + 1)
  \end{cases}
\end{align*}
\]

5. Introduce a new definition \( E \) for the restricted domain and to replace the dependent domain with \( E \):

\[
\begin{align*}
  & \kappa_5 = \text{add-def}(\kappa_4, 'E' = (D \times \text{interval}(0, u)) \text{ with } \lambda(u, i). (u \geq i))' \\
  & \kappa_6 = \text{subst}(\kappa_5, 'D \times \text{interval}(0, u)', 'E') \\
  & = \ 'P' = \lambda u : D.p(u, u) \\
  & \begin{cases}
    i = 0 \rightarrow \text{true} \\
    1 \leq i \leq u \rightarrow p(u, i - 1) \\
    \text{and } A(i) = A(u - i + 1)
  \end{cases}
\end{align*}
\]

6. Produce program \( p2 \):

\[
\text{produce-file}(p2, \text{def}(D), \text{def}(A), \kappa_6, \text{output-exp}(\rho)).
\]

Figure 4 shows the program \( p2 \) produced by the transformations. The meta program that transforms \( p1 \) into \( p2 \) is the concatenation of all the steps above. Figure 5 shows one parallel interpretation of \( p2 \). The new data field \( p \) is a two-dimensional data field defined over the trapezoidal domain \( E \), indexed by \((u, i)\). It computes the reduction linearly along the positive \( i \) direction. The input data field \( A \) is aligned with the column where \( u = n \). The data movements in \( p2 \) are somewhat more regular than \( p1 \).
5.3 Removing Broadcast

Since the value of $p(u, i)$ in p2 depends on the values of $A(i)$ and $A(u - i + 1)$, there are still long-distance data movements because each $A(i)$ is referenced by all $p(u, i)$ and $p(u, u - i + 1)$, $1 \leq i \leq n$. That is, values of $A(i)$ need to be broadcast to many other index points. Figure 5 shows how $A(1)$ is broadcast to all those index points referencing it. The broadcast can be eliminated by propagating $A(i)$ along the negative $u$ direction (starting from $u = n$) with a new data field $a$ defined over $E$. This can be done by the elim-broadcast operator. Elim-broadcast($\kappa, f, h, h$) looks in $\kappa$ for any usage of the data field $h$ in the definition for the data field $f$. Suppose the domain of $h$ is $D_h$ and the domain of $f$ is $D_f$, if $D_h$ is one dimension lower than $D_f$, then a new data field $h$ over $D_f$ is created to propagate the values of $h$ along the missing dimension, starting from the upper bound of that dimension. Simple induction over the missing dimension shows that the meaning of $f$ stays intact. Step 3 in the derivation below shows how elim-broadcast works.

1. Read in p2: $\rho = \text{parse-file}(p2)$.
2. Pick out the definition of $p$: $\kappa_1 = \text{def}(\lceil p \rceil, \rho)$.
3. Eliminate broadcasting of $A(i)$ and $A(u - i + 1)$ in $p$ by propagating values of $A$ with $a$ along the negative $u$ direction. Since for all $i, 1 \leq i \leq n$, we have $A(i) = A_i$.

\[
D = \text{interval}(1, n), \\
A = \lambda i : D. \text{read from input}, \\
P = \lambda u : D. \text{p}(u, u), \\
E = (D \times \text{interval}(0, n)) \langle (\lambda (u, i). (u \geq i)) \rangle, \\
p = \lambda (u, i) : E. \left\{ \\
\begin{array}{l}
1 \leq i \leq u \rightarrow \text{p}(u, i - 1) \\
\text{and } A(i) = A(u - i + 1)
\end{array} \right. \\
? P
\]

Figure 4: Program p2 with reduction replaced.

\[
D = \text{interval}(1, n), \\
E = (D \times \text{interval}(0, n)) \langle (\lambda (u, i). (u \geq i)) \rangle, \\
A = \lambda i : D. \text{read from input}, \\
P = \lambda u : D. \text{p}(u, u), \\
\begin{array}{l}
a = \lambda (u, i) : E. \left\{ \\
1 \leq i \leq u \rightarrow a(u + 1, i) \right. \\
\text{and } A(i) = A(u - i + 1)
\end{array} \\
\] ? P

Figure 6: Program p3 with broadcasting removed.

\[
a(u, i), 1 \leq u \leq n, \text{ the semantics of } p \text{ is preserved.}
\]

\[
\kappa_2 = \text{elim-broadcast}(\kappa_1, \lceil p \rceil, \lceil A \rceil, \lceil a \rceil)
\]

\[
\begin{array}{l}
\text{where} \\
\begin{array}{l}
a(u, i) = a(u, u - i + 1) \\
a(u, i) = A(i)
\end{array}
\end{array}
\]

4. Produce program p3:

\[
\text{produce-file}(p3, \text{def}(\lceil D \rceil, \rho), \text{def}(\lceil E \rceil, \rho), \text{def}(\lceil A \rceil, \rho), \text{def}(\lceil P \rceil, \rho), \text{flatten-local-def}(\kappa_2, \lceil p \rceil, \lceil a \rceil), \text{output-exp}(\rho)).
\]

Figure 6 shows the program p3 produced by the derivations above. The meta program that transforms p2 into p3 is the concatenation of all the steps above. Figure 7 shows one straightforward parallel interpretation of this program. The data fields $p$ and $a$ are placed over the same domain $E$, over which $p$ computes the reduction linearly along the positive $i$ direction and $a$ propagates values of $A$ along the negative $u$ direction. The input data field $A$ is fed into $a$ from the east side of $E$ (where $u = n$). In p3, values on each non-boundary index point $(u, i)$ of data fields $p$ and $a$ depend on the values on three other points: $(u + 1, i), (u, i - 1), (u, u - i + 1)$. The first two are neighborhood communications since they only access values on adjacent index points. The last one, $(u, u - i + 1)$, still requires long distance communication. However, these long distance communications are symmetric along the line $u = 2i - 1$, as shown by the dotted line in Figure 7.

5.4 Removing Symmetric Long Distance Communications

When a set of long distance communications has a symmetry point, line, or plane, we can fold the index domain
Figure 8: The data movements in p4 are all local.

along the symmetry so that the long distance communications are eliminated. We can eliminate the symmetric long distance communications by folding the domain $E$ according to the symmetry line with the following (restricted) domains and mappings:

$$E_1 = E | \{(\lambda(v,j) | (0 \leq j \leq (v+1)/2))\}$$
$$E_3 = E | \{(\lambda(v,j) | ((v+1)/2 < j \leq v))\}$$
$$E_2 = E_1 | \{(\lambda(v,j) | (j \neq 0 \land j \neq (v+1)/2))\}$$
$$E = \{(\lambda(v,j) | (0 < j < (v+1)/2))\}$$
$$g = \lambda(u,i) : E.$$
4. Distribute application into conditionals (i.e., \( f(e_1 \to e_2, e_3) = e_1 \to f(e_2, f(e_3)) \)), then simplify coproduct arrow with injections (i.e., \([a_1, a_2]((e_1)) = a_1(e) \) and \([a_1, a_2]((e_2)) = a_2(e)\)):

\[
\kappa_3 = \text{dist-app-}\text{-}\text{cond}(\kappa_3, '[a_1, a_2]')
\]

\[
\kappa_3 = \text{simplef-coprod-inj}(\kappa_3, '[a_1, a_2]')
\]

\[
= '[a_1, a_2] = \\
(\lambda(u, i) : E.
\begin{cases}
  u = n \to A(i), \\
  1 \leq u < n \to \\
  \begin{cases}
    0 \leq i \leq (u + 2)/2 \to \\
    a_1(u + 1, i) \\
    (u + 2)/2 < i \leq (u + 1) \to \\
    a_2(u + 1, u - i + 2)
  \end{cases}
\end{cases}
) \circ g_1,
\]

\[
\circ [g_1, g_2]'
\]

5. Distribute composition over coproduct arrow (i.e., \( f \circ [g_1, g_2] = [f \circ g_1, f \circ g_2] \)):

\[
\kappa_4 = \text{dist-comp-coproduct}(\kappa_3, '[g_1, g_2]')
\]

\[
= '[a_1, a_2] = \\
(\lambda(u, i) : E.
\begin{cases}
  u = n \to A(i), \\
  1 \leq u < n \to \\
  \begin{cases}
    0 \leq i \leq (u + 2)/2 \to \\
    a_1(u + 1, i) \\
    (u + 2)/2 < i \leq (u + 1) \to \\
    a_2(u + 1, u - i + 2)
  \end{cases}
\end{cases}
) \circ g_1
\]

\[
\circ [g_1, g_2]'
\]

6. Do \( \eta \)-abstraction over \( \kappa_4 \), which utilizes the unquoting facility. The selector lhs picks out the left hand side of a definition. Similarly, rhs picks out the right hand side. The selector coprod-1st picks out the first function of a coproduct arrow. Similarly for coprod-2nd.

\[
\kappa_5 = ['\text{lhs}(\kappa_4)] = \\
[\lambda(v, j) : E_1. (\text{coprod-1st}(\text{lhs}(\kappa_4)))(v, j)],
\]

\[
\lambda(v, j) : E_2. (\text{coprod-2nd}(\text{lhs}(\kappa_4)))(v, j)]'
\]

\[
= '[a_1, a_2] = \\
(\lambda(u, i) : E.
\begin{cases}
  u = n \to A(i), \\
  1 \leq u < n \to \\
  \begin{cases}
    0 \leq i \leq (u + 2)/2 \to \\
    a_1(u + 1, i) \\
    (u + 2)/2 < i \leq (u + 1) \to \\
    a_2(u + 1, u - i + 2)
  \end{cases}
\end{cases}
) \circ g_1
\]

\[
\circ [g_1, g_2]'
\]

7. Unfold composition; unfold \( g_1 \); unfold \( g_2 \); then normalize. Finally simplify the arithmetic:

\[
\kappa_5 = \text{unfold}(\text{unfold}(\kappa_5, 'o', '\rho'), 'g_1', '\rho'),
\]

\[
\kappa_6 = \text{simplef-}\text{-}\text{arith}(\text{normalize}(\text{unfold}(\kappa_5, 'g_1', '\rho')))
\]

\[
= '[a_1, a_2] = \\
(\lambda(v, j) : E_1.
\begin{cases}
  u = n \to A(i), \\
  1 \leq u < n \to \\
  \begin{cases}
    0 \leq i \leq (u + 2)/2 \to \\
    a_1(u + 1, i) \\
    (u + 2)/2 < i \leq (u + 1) \to \\
    a_2(u + 1, u - i + 2)
  \end{cases}
\end{cases}
) \circ g_1
\]

\[
\circ [g_1, g_2]'
\]

8. Eliminate dead code by simplifying the conditional expressions. For all \((v, j) \in E_1, (0 \leq j \leq (v + 2)/2)\) is always true and \((((v + 2)/2 < j \leq (v + 1))\) is always false. For all \((v, j) \in E_2, (0 \leq v - j + 1 \leq (v + 2)/2) \Rightarrow (j \leq v + 1 \land 2j \geq v) \Rightarrow (2j \geq v) \Rightarrow (2j = v) \) and \(((v + 2)/2 < v - j + 1 \leq (v + 1)) \Rightarrow (2j < v \land j \geq 0) \Rightarrow (2j < v)\). Therefore, the guards of the inner
Figure 9: Program p4 with distant symmetric communication removed.

The conditionals can be further simplified:

$$
\kappa = \text{simple-cond}(\kappa_0, \rho)
$$

where:

$$
\kappa_0 = \text{simple-cond}(\kappa_0, \rho)
$$

and properties of the coproduct arrow. The meta function contract(\(\sigma, \delta, \phi, \phi^{-1}, \rho\)), which corresponds closely to steps 3 to 6 above, properly replaces the occurrences of \(\sigma\) in \(\kappa\) with \(\delta\). It utilizes the equations \(\sigma = \delta \circ \phi\), \(\delta = \sigma \circ \phi^{-1}\), and properties of the coproduct arrow. The meta function contract(\(\sigma, \delta, \phi, \phi^{-1}, \rho\)), which corresponds to steps 2, 7, 8, and 9 above, derives \(\delta \equiv [a_1, a_2]\) from the definitions of \(\sigma, \phi, \phi^{-1}\) (all contained in \(\rho\)). It utilizes replace and \(\eta\)-abstraction. Using replace, the occurrence of \(p(u, u)\) in the definition of \(P\) of p3 can be replaced by \(p_1\) and \(p_2\), as shown in the program p4 in Figure 9.

Using the ideas presented above, the meta program dc in Figure 10 transforms p3 into the new program p4 in Figure 9. Compared with p3, p4 has extra conditional expressions. They take care of the boundary conditions on and near the symmetry line where the values of \(a_1\) and \(p_1\) are "passed onto" \(a_2\) and \(p_2\). Program p4 also has local communication because for any index point \((v, j)\) only the values on adjacent points \((v, j-1), (v, j+1), (v+1, j),\) and \((v+1, j+1)\) are needed. It is much easier for the compiler to produce efficient code because, firstly, mapping the two-dimensional index domain \(E_{1} + E_{2}\) onto a hypercube or onto a two-dimensional mesh is very straightforward, and, secondly, the communication now are all localized.
and there is no need to minimize long distance communications.

Figure 8 shows the data movements in p4. The computation takes place on the index domain $E_1 + E_2$. The data field $p_1$ (defined over $E_1$) computes the reduction along the positive i direction (as p does in p3) until it reaches the symmetry line at which point the partial result is passed to $p_2$ (defined over $E_2$), which continues to compute the reduction along the negative i direction. The data field $a_1$ (defined over $E_1$) propagates the values of $A(i), 1 \leq i \leq \lfloor n/2 \rfloor$, along the negative u direction until it reaches the symmetry line, where the value is passed to $a_2$ and propagated diagonally. The data field $a_2$ (defined over $E_2$) propagates the values of $A(i), \lfloor n/2 \rfloor + 1 \leq i \leq n$ in the south-west direction.

6 Related Works

The notion of quoting and unquoting has been in use in some languages for a long time. However, they come in different variations and lack a formal treatment. The language that is most similar in spirit to our work is Brian Smith's quote, up arrow, and down arrow constructs in his 3-Lisp [18, 19]. The language 3-Lisp is designed for expressing reflection in Lisp. Compared to 3-Lisp, our metaconstructs are conceptually simpler and easier to comprehend. The goal of 3-Lisp is very different from ours. For our goal of transforming parallel-programs, our metalinguistic features are adequate.

Our quoting and unquoting are reminiscent of the backquote and comma of many Lisp dialects, e.g. the CommonLisp [20]. Though very similar in appearance, the semantics of quoting and unquoting between our work and Lisp are different. Quoting an S-expression in Lisp prevents the eval from evaluating the S-expression and the quoting returns the S-expression itself, not the structure of the S-expression. Though one may argue that the S-expression is the structure for itself, quoting an atom in Lisp returns the atom itself, which lacks the properties of S-expressions that make them seem self-representing. If quoting in Lisp were to be viewed as returning the structures, then, for example, (+ '1 '2) should not have been evaluated to 3 because + should not work on structures. The concepts of notation and structure are somewhat confused in the semantics of Lisp.

There are a few existing languages, such as ML [16], that are equipped with features for user-defined data types. Structures can be represented with user-defined data type and the operators can be implemented as functions over this type. However, for our application we feel that structures are very fundamental semantic objects. Their importance justifies direct notations such as quoting and unquoting for them. The benefit of having direct notation is that programmers can perceive and manipulate structures at the closest conceptual level without having to encode and decode structures to and from indirect representations in terms of some user-defined data type. This is analogous to that a language aiming at numerical computation should have direct notations for floating point numbers rather than ask users to encode them with a pair of integers representing mantissas and exponents.

Many program transformation systems have been reported for transforming traditional sequential programs. For example, [8, 3, 9, 17, 15, 21, 11, 2]. Most program transformation systems have put their emphasis on automatic program transformations based on some fixed set of transformation strategies. Few of them provide users with facilities to express their own transformation strategies. A fixed set of strategies employed in an automatic transformation system may perform well for the programs with targeted properties. However, any fixed set of strategies is unlikely to work well in general. We believe that, in addition to the system default transformation strategies, providing users with a metalanguage to express their own transformation strategies for special cases is more desirable. The work that is closest to our philosophy is the Munich CIP project [2]. However, their metalanguage, which has only a few simple constructs such as union, product, closure, is not as general as what is presented in this paper. Also, using equations, η-abstraction, and normalization in our transformation leads us to new programs that can not be obtained by the traditional fold unfold transformations.

7 Concluding Remarks

We have described a set of metalinguistic features which include a rationalized version of quoting and unquoting and a set of constructors, selectors, predicates and operators on structures. Using these, we have demonstrated how a class of parallel-programs for solving palindrome recognition problem are formally derived. All the transformation steps preserve the semantics of the initial program. Therefore, the transformation process itself can be viewed as a proof of the equivalence among all derived programs.

The philosophy motivating this work can be summarized as follows: since program transformation is an effective methodology for deriving more efficient programs and no parallelizing compiler is able to perform well for all input programs, we feel essential metalinguistic features which allows the user to express their transformation schemes at a high conceptual level should be very useful. This metalinguistic features should be practical, easy-to-use, and be implemented without much effort. We feel that the presented features satisfies these requirements quite well.

An interactive environment similar to a Lisp interpreter has been built for Crystal extended with these metalinguistic features. 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. The metalinguistic features presented (quoting, unquoting, and various semantics preserving operators on structures) are applicable to other lan-
guages for building their own program manipulation and transformation systems.

References


