Yale University
Department of Computer Science

Meta-Crystal – A Metalanguage for Parallel-Program Optimization

J. Allan Yang   Young-il Choo

YALEU/DCS/TR-786
April 1990

This work was supported in part by the ONR grant N00014-89-J-1906.

Meta-Crystal – A Metalanguage for Parallel-Program Optimization

J. Allan Yang　　Young-il Choo

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

Abstract

This paper describes the design and implementation of Meta-Crystal, a metalanguage for doing parallel-program optimization by algebraic program transformation with domain morphisms. The purpose of Meta-Crystal is to provide users with full control over how their programs are to be transformed. Meta-Crystal takes care of the tedious mechanical and algebraic manipulation and allow users to work at a higher conceptual level. The key features of Meta-Crystal are its quoting and unquoting constructs which provide an elegant way to build and manipulate Crystal structures. An implementation of Meta-Crystal is also sketched.

Key words: parallel-program optimization, algebraic program transformation, metalanguage.

1 Introduction

Since efficiency and readability of programs are often conflicting goals, program transformation has been used to transform readable yet inefficient programs into efficient ones ([Darlington 81], [Darlington & Burstall 76], [Burstall & Darlington 77], [Feather 82], [Loveman 77], [Reddy 88]). However, most program transformation systems only provide *pre-packaged* functionalities. Users of such systems have very little control over the transformations applied to their programs. In order to allow users to express their own transformations, we need a meta language that allows users to directly access program text of the object language and provide users with a collection of primitives for constructing and manipulating the program.

As various parallel machines have become available in the late 1980's, two major approaches to harnessing parallelism have been building parallelizing compilers and programming directly with parallel constructs. Parallelizing compilers have a similar problem to traditional program transformation systems: users have no control over the optimization strategies applied. Explicit programming for parallelism puts the burden of managing efficiency and readability onto users. Except for programs that are embarrassingly parallel, it is often difficult for programmers to manage the tradeoffs between the cost of synchronization, communication, and load balance of the target program. The situation gets worse when the size of machines scales up and when the memory becomes distributed or has a hierarchical structure.

The approach taken by Crystal [Chen 86b] for programming parallel machines is to start out with a simple, intuitive, yet maybe inefficient, program, and to use source to source program transformation to optimize it [Chen 86a]. To provide users with the control over the program transformations in addition to compiler's default strategies, a metalanguage, Meta-Crystal, has been designed and implemented. This report describes the issues of the design and implementation of Meta-Crystal. The main purpose of the metalanguage is to automate the tedious and error-prone aspects of program transformation and to provide users with direct control over the optimizations applied to their programs. Some of the known techniques, such as fan-reduction, domain contraction, and space-time mappings [Chen 86a] (collectively known as *reshape* and *refinement* domain morphisms [Choo & Chen 88]), can be expressed as procedures in Meta-Crystal . These techniques serve as default optimization strategies when users are willing to let the Crystal compiler do the transformations.

Because λ-notation is used in Crystal for function declarations and some properties of λ-calculus [Church 41] are used in Meta-Crystal for transforming programs, we provide a brief introduction to the λ-notation in this paragraph for readers who are not familiar with it. 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 "."). 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 $f3$ *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 $\lambda x.(fx)$. That is, f is equivalent to $\lambda x.(fx)$ by η-abstraction.

This paper is organized as follows. Section 2 describes the essential features of Meta-Crystal: quoting, unquoting, constructors, selectors, predicates, and operators. Section 3 contains an example showing how Meta-Crystal can be used to optimize parallel-programs by algebraic program transformations. Section 4 sketches the implementation of Meta-Crystal. The organization of Meta-Crystal processor, the basic data structure, and the algorithms used to implement operators, quoting, and unquoting are described. Section 5 summaries this report and discusses possible extensions.

2 The Design of Meta-Crystal

A Crystal program consists of a set of function declarations. Function declarations are considered as equations. The purpose of Meta-Crystal is to provide users with a way of doing algebraic program transformations to optimize their programs. The transformation done at each step must preserve the validity of the derived equation. For a simple example, if a Crystal program contains $f = \lambda x.x + 1$ and $g = f3$, another valid equation $g = 3 + 1$ can be derived.

Meta-Crystal is designed to be a simple functional language with lexical scoping. We first discuss the basic notions and essential features of Meta-Crystal.

2.1 Notations and Structures

We call Crystal the *object* language and Meta-Crystal the *meta* language. We use "expressions of the object language" interchangeably with "expressions at the object level". In order to allow users to manipulate object level programs, we need to treat object level equations and expressions as first class objects at the meta level. To this end, we need to introduce the following two notions.

First, we call a syntactically correct sequence of characters that represents a program, an equation, or an expression a *notation.* For clarity of discussion in this section, we will use the typewriter font for notations. For example, the expressions `(λx.x)` and `3` are notations. Notations are composed of characters from the alphabet of the language of interest.

Second, there are *structures* that correspond to notations. We use the slanted font for structures. For example, the structure of notation `(λx.x+1)3` is *(λx.x+1)3.* Structures can be thought of as abstract syntax trees. Structures enjoy some algebraic properties and have a set of constructors, selectors, predicates, and operators associated with them. 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), which is *λx.x+1*; and its rand (i.e., operands), which is *3*. We also have an operator **normalize** for reducing β-redexes and transforming *(λx.x+1)3* to *3+1*. We will use "object level structures" equivalently to "structures of object level notations" in the later discussions.

Provided that the language of interest is unambiguous, there is a unique structure for every notation. For languages that are not concerned about manipulating programs, the distinction between notation and structure is not important because structures need not be accessible to users. When we are interested in constructing and manipulating programs, we want structures to be first class objects at the meta level. That is, structures should be *expressible* (i.e., expressions can produce structures as their result) as well as *denotable* (i.e., structures can be bound to identifiers in the environment) in the meta language.

Please note that there could be more than one notation corresponding to a particular structure. For example, *3* could be the structure of `3`, `(3)`, `((3))`, etc. But these notations should all denote the same structure, and we can pick any of these to be the notation of the structure *3*. Therefore, when we say "the notation of the structure x", we mean any notation that has x as its structure. When there is no danger of confusion, we will use the terms "notation" and "structure" equivalently.

2.2 Essential Features of Meta-Crystal

We now discuss the features of Meta-Crystal for notating and manipulating Crystal structures.

Quoting

To express Crystal structures in Meta-Crystal, we use Quine's quasi-quoting mechanism. It is notated by putting `‘` and `’` around a Crystal notation. We only allow notations for Crystal programs, equations, and expressions to be quoted. The exact syntax for these Crystal notations is given by the Crystal grammar.

Let φ be any Crystal notation allowed inside quoting, ‘φ’ denotes the structure corresponding to φ. In summary, ‘φ’ the meta level expression for denoting the structure of the Crystal notation φ (without being normalized):

‘φ’ denotes the structure of the notation φ.

Unquoting

In order to provide convenient abstraction over structures, we need unquoting abilities inside quoted Crystal expressions. We use the notation $[\![$ and $]\!]$ around a *meta* expression to notate the unquoting. Unquoted meta expressions can only appear inside a quoted object expression.

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,

$$`\ldots\ [\![\tau]\!]\ \ldots' \quad \text{denotes} \quad `\ldots\ \psi \ldots',$$

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.

For a simple example, for any Crystal notation φ,

$$`\ldots\ [\![`\varphi']\!]\ \ldots' \ = `\ldots\ \varphi \ldots'$$

because $`\varphi'$ is a constant at the meta level and therefore $[\![`\varphi']\!]$ evaluates to φ.

Because we always use notations to express both Meta-Crystal and Crystal definitions and expressions, we will no longer insist on using typewriter font for notations. For more concise presentation, we will use mathematical fonts to express both Meta-Crystal and Crystal notations. Greek letters such as $\alpha, \beta, \kappa, \tau$ and ϕ are used for meta variables having Crystal structures as their bindings. Meta-Crystal functions and program constructs will be written in sans serif style.

For another example on unquoting, given the following meta definitions

$$\begin{aligned} \alpha &= `(\lambda x.x)3' \\ \beta &= \mathsf{normalize}(\alpha) \\ f &= \lambda \mathsf{x}.`[\![\mathsf{x}]\!] + [\![\mathsf{x}]\!]' \quad , \end{aligned}$$

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

$$\begin{aligned} f(`1') &= `1 + 1' \\ f(\alpha) &= `(\lambda x.x)3 + (\lambda x.x)3' \\ f(\beta) &= `3 + 3' \quad . \end{aligned}$$

Notice that an unquoted meta expression is only meaningful inside a quoted Crystal expression and it must evaluate to a Crystal structure.

Meta Environments

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 object expression. For example, given the meta definitions

$$\begin{aligned} \alpha &= `1' \\ f &= \lambda x.`(\lambda x.[\![x]\!] + [\![\alpha]\!])A' \quad , \end{aligned}$$

the unquoted x in the definition of f is bound to the first occurrence of x, the formal argument of the meta function. And we have

$$\begin{aligned}
&f(\text{`}x\text{'}) = \text{`}(\lambda x.x+1)A\text{'}\\
&\mathsf{normalize}(f(\text{`}x\text{'})) = \text{`}A+1\text{'}\\
&f(\text{`}y\text{'}) = \text{`}(\lambda x.y+1)A\text{'}\\
&\mathsf{normalize}(f(\text{`}y\text{'})) = \text{`}y+1\text{'} \quad .
\end{aligned}$$

Constructors, Selectors, and Predicates

In Meta-Crystal a set of constructors, selectors, and predicates is provided for each Crystal structure. For example, for the structure of application we have the following:

$$\begin{aligned}
&\mathsf{mk\text{-}appl}(\tau_1,\tau_2) = \text{`}[\![\tau_1]\!]([\![\tau_2]\!])\text{'}\\
&\mathsf{rator}(\mathsf{mk\text{-}appl}(\tau_1,\tau_2)) = \tau_1\\
&\mathsf{rand}(\mathsf{mk\text{-}appl}(\tau_1,\tau_2)) = \tau_2\\
&\mathsf{appl?}(\tau) = \begin{cases} \mathsf{true} & \text{if } \tau = \mathsf{mk\text{-}appl}(\tau_1,\tau_2) \text{ for some } \tau_1,\tau_2\\ \mathsf{false} & \text{otherwise.}\end{cases}
\end{aligned}$$

Operators

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. Given the meta definitions

$$\begin{aligned}
&\kappa = \text{`}f \circ ((\lambda f.fx)g)\text{'}\\
&\phi = \text{`}h\text{'} \quad ,
\end{aligned}$$

we have

$$\begin{aligned}
&\mathsf{subst}(\kappa, \text{`}f\text{'}, \phi) = \text{`}h \circ ((\lambda f.fx)g)\text{'}\\
&\mathsf{normalize}(\kappa) = \text{`}f \circ (gx)\text{'} \quad .
\end{aligned}$$

Let κ range over equations, τ over terms, ϕ over function names, and ρ over sets of equations, some of the other operators in Meta-Crystal are defined below. The operators **subst**, **unfold** and **simplify-arith** will be used in the example to be discussed in the next section.

parse-file(`file`) Denotes the set of equations contained in `file`.

def(ϕ, ρ) Denotes the equation, which is contained in ρ, that defines ϕ.

reduce(τ) β-reduces the β-redex τ.

expand(κ, ϕ, ρ) Substitutes all occurrence of ϕ in κ with the right hand side of the equation in ρ defining ϕ.

unfold(κ, ϕ, ρ) replaces the function (defined in ρ) named by ϕ and its argument with its body with appropriate substitutions of the arguments in the equation κ. Note that this is equivalent to **expand** of the function followed by a **reduce** or **normalize**.

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

Procedural Abstraction

Meta-Crystal provides users with procedural abstraction for defining their own transformation schemes. We adopt the λ-notation to represent functions in Meta-Crystal. For example, a meta function max that takes a pair (α, β) and returns the structure '$\textsf{if } [\![\alpha]\!] > [\![\beta]\!] \textsf{ then } [\![\alpha]\!] \textsf{ else } [\![\beta]\!] \textsf{ fi}$' is expressed as

$$\textsf{max} \;=\; \lambda(\alpha,\beta).\ \text{'}\textsf{if } [\![\alpha]\!] > [\![\beta]\!] \textsf{ then } [\![\alpha]\!] \textsf{ else } [\![\beta]\!] \textsf{ fi}\text{'} \quad .$$

Then $\textsf{max}(\text{'}1+x\text{'}, \text{'}2\text{'})$ returns the structure '$\textsf{if } (1+x) > 2 \textsf{ then } (1+x) \textsf{ else } 2 \textsf{ fi}$'. This has the benefit of simplicity in syntax, and allows us to merge Crystal and Meta-Crystal cleanly in the future.

3 An Example: Reshape Morphism

In this section we show how Meta-Crystal is used in doing algebraic program transformation to optimize parallel-programs. Consider the following Crystal program which computes the matrix product, $A \times B$, in a three dimensional space by propagating A along the j-dimension with a, B along i-dimension with b, and by accumulating the sum along the third k-dimension with c:

$$\begin{aligned}
D &= 0\,..\,n \\
E &= 0\,..\,2n-1 \\
T &= 0\,..\,3n-2 \\
a &= \lambda(i,j,k)\colon D^3.\left\{\begin{array}{l} j=0 \rightarrow A(i,k) \\ 0<j\leq n \rightarrow a(i,j-1,k) \end{array}\right\} \\
b &= \lambda(i,j,k)\colon D^3.\left\{\begin{array}{l} i=0 \rightarrow B(k,j) \\ 0<i\leq n \rightarrow b(i-1,j,k) \end{array}\right\} \\
c &= \lambda(i,j,k)\colon D^3.\left\{\begin{array}{l} k=0 \rightarrow 0 \\ 0<k\leq n \rightarrow a(i,j,k)\times b(i,j,k)+c(i,j,k-1) \end{array}\right\}
\end{aligned}$$

In this program, D is an interval from 0 to n, similarly for E, and T. We call D, E, and T *index domains*. D^3 is another index domain constructed from $D \times D \times D$ where $\times$ is the usual cartesian product. The definition of a says that it is a function that takes a tuple of three indices, (i, j, k), of type D^3 (we use ":" to tag a type to an expression) as input argument, and returns $A(i, k)$ if j is 0, or $a(i, j-1, k)$ otherwise. Note that alternative choices of a conditional expression are enclosed by "{" and "}" and each branch has the form "condition $\rightarrow$ consequence". We call a, b, and c *data fields*. Data fields map points of an index domain to values. The input matrix are A and B. The value of matrix A at row i, column j is written as $A(i, j)$, similarly for B.

A direct implementation of this program uses space of size n^3 (the cardinality of D^3) and takes roughly $2n$ steps to complete the computation. At each step, only a few points in D^3 are active. The space utilization of this implementation is not very efficient. There are many ways to optimize this program. Consider the following linear mappings:

$$\begin{aligned}
g &= \lambda(i,j,k)\colon D^3.(i+k, j+k, i+j+k) : E^2 \times T \\
g^{-1} &= \lambda(i,j,k)\colon E^2 \times T.(k-j, k-i, i+j-k) : D^3
\end{aligned}$$

The function g maps points in D^3 to points in $E^2 \times T$ and g^{-1} does the inverse. One direct implementation of $E^2 \times T$ uses space of size $4n^2$ (the cardinality of E^2) and takes $3n - 1$ (the cardinality of T) steps, thereby we achieve a much better space utilization. Coming up with these mappings requires some insight. Deriving a new program by hand with these mappings is tedious and error-prone. Meta-Crystal can assist in deriving a new program with algebraic program transformations. 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.

The task can be generalized as follows. Let D_1, D_2 be two index domains, and $a : D_1 \rightarrow V$, $\hat{a} : D_2 \rightarrow V$ be two data fields. The mapping $g : D_1 \rightarrow D_2$ is a *reshape morphism* if it has an inverse $g^{-1} : D_2 \rightarrow D_1$ which makes the following diagram commute:

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

Our goal is to derive $\hat{a}$ given a, g, and g^{-1} in the case that computing the program for $\hat{a}$ is less expensive than that for a. A canonical procedure for deriving $\hat{a}$ by utilizing the equality $\hat{a} = a \circ g^{-1}$, unfolding of g and g^{-1}, and some equational transformations is developed in [Choo & Chen 88]. A simplified version using similar ideas is presented in the following derivation for $\hat{a}$.

Let the set of equations defining a, b, c, g and g^{-1} are contained in the object program environment denoted by the meta variable ρ, we can derive $\hat{a}$ as below:

1. Do η-abstraction[1] on $a \circ g^{-1}$ in the equation '$\hat{a} = a \circ g^{-1}$', which is valid by definition:

$$\kappa_1 = \text{`}\hat{a} = \lambda(i,j,k) : E^2 \times T.(a \circ g^{-1}(i,j,k))\text{'}$$

2. Unfold the composition:

$$\begin{aligned}\kappa_2 &= \mathsf{unfold}(\kappa_1, \text{`} \circ \text{'}, \rho) \\ &= \text{`}\hat{a} = \lambda(i,j,k) : E^2 \times T.a(g^{-1}(i,j,k))\text{'}\end{aligned}$$

3. Unfold g^{-1}:

$$\begin{aligned}\kappa_3 &= \mathsf{unfold}(\kappa_2, \text{`}g^{-1}\text{'}, \rho) \\ &= \text{`}\hat{a} = \lambda(i,j,k) : E^2 \times T.a(k-j, k-i, i+j-k)\text{'}\end{aligned}$$

4. Unfold a:

$$\begin{aligned}\kappa_4 &= \mathsf{unfold}(\kappa_3, \text{`}a\text{'}, \rho) \\ &= \text{`}\hat{a} = \lambda(i,j,k) : E^2 \times T.\left\{\begin{array}{l} k-j=0 \rightarrow A(k-j, i+j-k) \\ 0 < k-j \leq n \rightarrow a(k-j, k-i-1, i+j-k) \end{array}\right\}\text{'},\end{aligned}$$

[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.

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

$$\begin{aligned}\kappa_5 &= \mathsf{subst}(\kappa_4, `a', `\hat{a} \circ g') \\ &= `\hat{a} = \lambda(i,j,k) : E^2 \times T. \left\{ \begin{array}{l} k-j=0 \rightarrow A(k-j, i+j-k) \\ 0 < k-j \le n \rightarrow (\hat{a} \circ g)(k-j, k-i-1, i+j-k) \end{array} \right\}'\end{aligned}$$

6. Unfold composition, then unfold g:

$$\begin{aligned}\kappa_6 &= \mathsf{unfold}(\mathsf{unfold}(\kappa_5, ` \circ ', \rho), `g', \rho) \\ &= `\hat{a} = \lambda(i,j,k) : E^2 \times T. \left\{ \begin{array}{l} k-j=0 \rightarrow A(k-j, i+j-k) \\ 0 < k-j \le n \rightarrow \hat{a}(k-j+i+j-k, \\ \quad k-i-1+i+j-k, k-j+k-i-1+i+j-k) \end{array} \right\}'\end{aligned}$$

7. Finally, simplify the arithmetics:

$$\begin{aligned}\kappa_7 &= \mathsf{simplify\text{-}arith}(\kappa_6) \\ &= `\hat{a} = \lambda(i,j,k) : E^2 \times T. \left\{ \begin{array}{l} k=j \rightarrow A(k-j, i+j-k) \\ 0 < k-j \le n \rightarrow \hat{a}(i, j-1, k-i) \end{array} \right\}'\end{aligned}$$

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**:

$$\begin{aligned}\mathsf{reshape} = \lambda(\alpha, \hat{\alpha}, \phi, \phi^{-1}, \rho).\ \gamma \quad \mathsf{where}\{ \\ &\kappa_1 = `[\![\hat{\alpha}]\!] = \lambda(i,j,k) : E^2 \times T.(([\![\alpha]\!] \circ [\![\phi^{-1}]\!])(i,j,k))' \\ &\kappa_2 = \mathsf{unfold}(\kappa_1, ` \circ ', \rho) \\ &\kappa_3 = \mathsf{unfold}(\kappa_2, \phi^{-1}, \rho) \\ &\kappa_4 = \mathsf{unfold}(\kappa_3, \alpha, \rho) \\ &\kappa_5 = \mathsf{subst}(\kappa_4, \alpha, `[\![\hat{\alpha}]\!] \circ [\![\phi]\!]') \\ &\kappa_6 = \mathsf{unfold}(\mathsf{unfold}(\kappa_5, ` \circ ', \rho), \phi, \rho) \\ &\gamma = \mathsf{simplify\text{-}arith}(\kappa_6) \\ \}\end{aligned}$$

Using **reshape**, $\hat{a}$ can be derived by **reshape**($`a'$, $`\hat{a}'$, $`g'$, $`g^{-1}'$, ρ); $\hat{b}$ can be derived by **reshape** ($`b'$, $`\hat{b}'$, $`g'$, $`g^{-1}'$, ρ); and $\hat{c}$ can be derived by **reshape**($`c'$, $`\hat{c}'$, $`g'$, $`g^{-1}'$, ρ), followed by a few more steps to replace occurrences of a and b with $\hat{a}$ and $\hat{b}$. The derived optimized program is:

$$\begin{aligned}E &= 0 \mathinner{..} 2n-1 \\ T &= 0 \mathinner{..} 3n-2 \\ \hat{a} &= \lambda(x,y,t) : E^2 \times T. \left\{ \begin{array}{l} t-x=0 \rightarrow A(t-y, x+y-t) \\ 0 < t-x \le n \rightarrow \hat{a}(x, y-1, t-1) \end{array} \right\}\end{aligned}$$

$$\hat{b} = \lambda(x,y,t): E^2 \times T. \left\{ \begin{array}{l} t-y=0 \rightarrow B(t-y, x+y-t) \\ 0 < t-y \leq n \rightarrow \hat{b}(x-1,y,t-1) \end{array} \right\}$$

$$\hat{c} = \lambda(x,y,t): E^2 * T. \left\{ \begin{array}{l} x+y-t=0 \rightarrow 0 \\ 0 < x+y-t \leq n \rightarrow \hat{c}(x-1,y-1,t-1)+ \\ \quad \hat{a}(x,y,t) \times \hat{b}(x,y,t) \end{array} \right\} .$$

This derived program is similar to a systolic matrix multiplication algorithm presented in [Kung & Leiserson 80].

4 Implementation of Meta-Crystal

This section describes the implementation of Meta-Crystal. Figure 1 shows the overall structure of Meta-Crystal.

The language T (a dialect of Lisp) is used to implement Meta-Crystal. The lexer scans the input character strings and returns one token only on the request of the parser. It is a lazy lexer. The parser is generated by *tyacc*, a T version of *yacc* [Aho & Johnson 74]. It transforms the token stream into Meta-Crystal parse nodes, which are T lists. To minimize the effort for developing a working prototype in a short time, T is used as the underlying machinery for executing Meta-Crystal programs. A translator is used to translate Meta-Crystal parse nodes into equivalent T expressions using syntax directed translation. The execution of Meta-Crystal programs is handled by the `eval` of T.

The heart of the system lies in the run-time support library, which is a collection of T procedures for constructing and transforming Crystal structures. The library contains the following:

1. A specialized Crystal lexer and parser that provides Meta-Crystal with Crystal structures.
2. The constructor, selector, and predicate for each Crystal structure.
3. Operators that transform Crystal structures.
4. The quoting and the unquoting mechanisms.
5. A pretty printer for printing out Crystal structures in Crystal syntax.

We will briefly sketch the implementation of the constructors, selectors, predicates, operators, and the quoting and the unquoting mechanisms.

Representation of Crystal Structures

Crystal structures (i.e., parse trees) are represented as lists of the following format:

```
(type f1 ... fn)
```

where `type` is the type of the structure and `f1` ... `fn` are the fields. A field may in turn be another Crystal structure. For example, an identifier `x` is represented as `(*id* x)`; a function application `f (if p then x else y fi)` is represented as

```
(*appl* (*id* f) (*cond* ((*id p) (*id x)) (else (*id* y)))),
```

the first field is the rator, the second field is the rand.

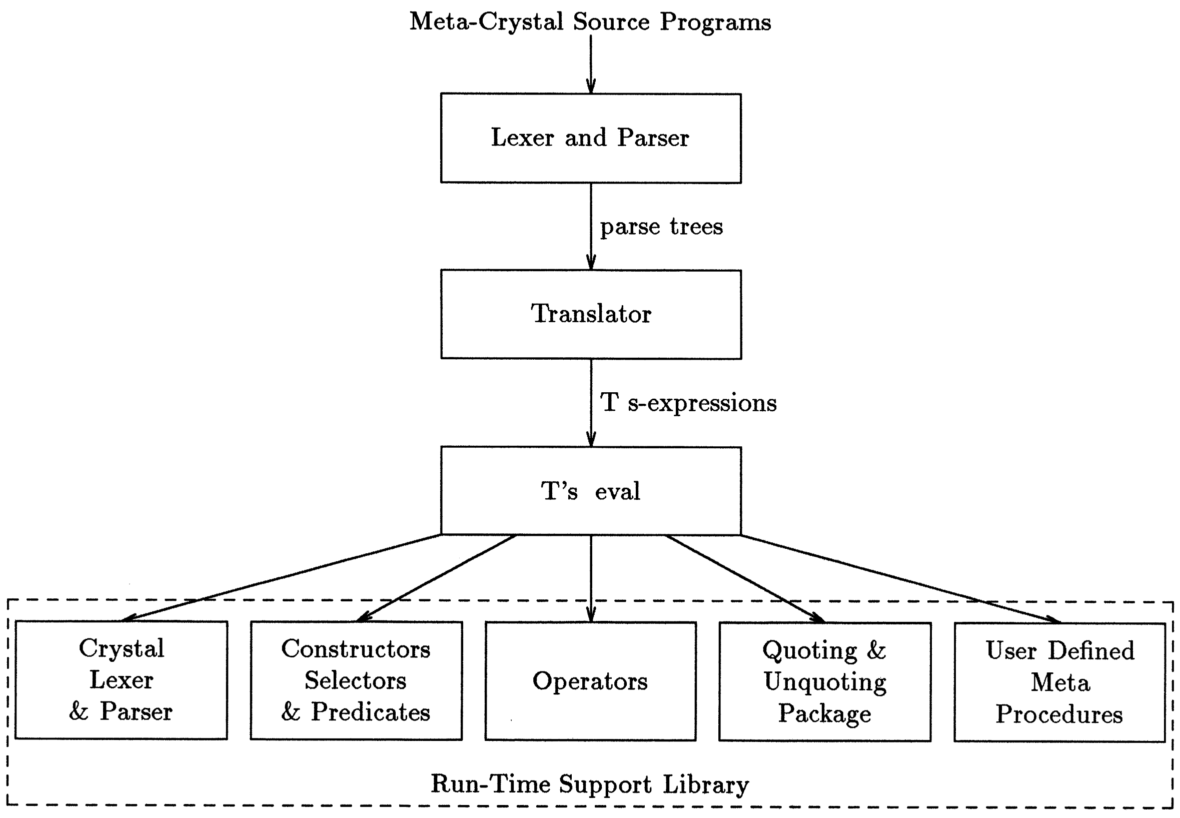
Meta-Crystal Source Programs
Lexer and Parser
parse trees
Translator
T s-expressions
T's eval
Crystal Lexer & Parser
Constructors Selectors & Predicates
Operators
Quoting & Unquoting Package
User Defined Meta Procedures
Run-Time Support Library

Figure 1: The structure of Meta-Crystal processor

Implementing Constructors, Selectors, and Predicates

Given the representation of Crystal structures described above, implementing constructors, selectors, and predicates is quite simple. A constructor is simply a procedure that constructs a list with the proper type. A selector is a procedure for selecting a particular field of a list of certain type. A predicate is a procedure for testing whether a list is of the right type. For example, the constructor, selectors, and predicate of structure `*appl*` are as below:

```
(define (mk-appl rator rand) (list '*appl* rator rand))
(define (appl? node) (eq? (car node) '*appl*))
(define (rator node)
    (if (appl? node) (cadr node) (error "not an appl node")))
(define (rand node)
    (if (appl? node) (caddr node) (error "not an appl node")))
```

Implementing Operators

The operators transform Crystal structures. These operators will have to work on lists, or equivalently, on trees. One common pattern of operation on trees is to find a subtree that satisfies some property, and then apply some function to that subtree:

```
;; Find a subtree in Tree that satisfies predicate P,
;; and apply function F to that subtree.
(define (transform-tree Tree Pred? Fun)
    (cond
     ((or (null? Tree) (atom? Tree)) Tree)
     ((Pred? Tree) (Fun Tree))
     (else (cons (transform-tree (car Tree) Pred? Fun)
                 (transform-tree (cdr Tree) Pred? Fun)))))
```

For example, a simplified version of **subst**(κ, τ_1, τ_2) can be implemented as:

```
(define (subst k t1 t2)
    (transform-tree k
            (lambda (t) (alikev? t t1))
            (lambda (t) t2)))
```

Extra checking is needed if τ_1 is an identifier to make sure that **subst** does not replace bound occurrences of τ_1. Take **normalize** for another example, we want to look for a subtree of type application with a rator of type abstraction and then substitute the formals with actuals. Referring to the definitions of operators in section 2, we observe that all operators can be implemented in this fashion except **simplify-arith** and **def**. Because of the top down nature of the algorithm, **normalize** does normal order reduction which eventually will reach the normal form if there is one. Care must be taken to respect lexical scoping of bound variables in lambda terms when doing substitution in **expand**, **reduce**, **normalize** and **unfold**.

Assuming the predicate `Pred?` and function `Fun` take constant time to evaluate, the complexity of `transform-tree` is of $O(n)$, where n is the size of `Tree`. In practice the assumptions are usually true.

Implementing Quoting And Unquoting

Let's take `'f(⟦x⟧)'` for example. The lexer picks up a q-quoted string (with or without unquoting) as a single token: `(Token QQSTRING "f(⟦x⟧)")`. The parser builds a special parse node for it: `(*qqstring* "f(⟦x⟧)")`. And the translator translates it into a T expression: `(process-qqstring "f(⟦x⟧)")`.

`Process-qqstring` is a macro that, when evaluated, results in a T expression which does the following:

1. Substitutes all unquoted meta expressions (i.e., ⟦x⟧ in this example) with unique unused Crystal identifiers as place holders (e.g., `%id-1`).
2. Calls the Crystal lexer and parser to translate the string (i.e., `f(%id-1)` in this example) into a Crystal structure.
3. Calls the meta lexer, parser, and translator to translate the unquoted meta expression (i.e., `x` in this example) into a T expression and evaluate it in the most current environment. The result should be a Crystal structure.
4. Substitutes the inserted unique Crystal identifiers in the result of step 2 (i.e., `%id-1` in this example) with its corresponding result produced at step 3.

An interesting point to notice is that the dynamic scoping (with macro) of T is used to achieve the lexical scoping in Meta-Crystal. Arbitrary levels of nested quoting and unquoting is supported in this implementation.

5 Summary and Possible Extensions

This paper on Meta-Crystal can be summarized as follows:

- Meta-Crystal does algebraic program transformation based on domain morphisms, equational substitutions, and properties of the λ-calculus.
- The metalanguage takes care of the mechanical algebraic manipulation.
- Programmers can start with simple initial object programs.
- Desired properties are obtained in the derived object program.
- All transformations are formal.
- Users have full control over transformations.

Some possible extensions along this line are as follows. *Unifying the object and the meta languages* has the benefit of slimmer implementation since only one set of lexer and parser is needed instead of two. However, this will introduce other language issues such as whether or not we should add a different quoting with normalization and an unquoting without normalization? How should the bindings of free variables be dealt with when quoting an expression with free variables? Similar problem arises in quoting function definitions with free variables. Must environments become first class also?

Making programs first class object is applicable to any language which needs to process programs. Partial evaluation is one good example. Annotation is one way to express user's control about when to partial evaluate a function invocation. A metalanguage equipped with quoting, unquoting, and suitable operators serves the same purpose, and is perhaps more expressive than annotation. Another example is compilation. A compiler can be viewed as a huge and complicated meta program that transforms programs. Most of the time the only control provided to users are perhaps some switches. This approach may be all right for compiling sequential-programs. For compiling parallel-programs finer grained user-controllability is desired because of the huge space of tradeoffs.

Acknowledgement

The theoretical groundwork developed by Young-il Choo and Marina Chen in [Choo & Chen 88] is fundamental for this work. Thanks also goes to Michel Jacquemin for his Crystal parser and many helpful discussions during implementation.

References

[Aho & Johnson 74] A. V. Aho and S. C. Johnson. Programming utilities and libraries: LR parsing. *Computing Serveys*, June 1974.

[Burstall & Darlington 77] 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.

[Chen 86a] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[Chen 86b] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[Choo & Chen 88] 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.

[Church 41] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1941.

[Darlington & Burstall 76] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

[Darlington 81] John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16:1–46, 1981.

[Feather 82] Martin S. Feather. A system for assisting program transformation. *ACM Transaction on Programming Language and Systems*, pages 1–20, January 1982.

[Kung & Leiserson 80] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. In *Introduction to VLSI Systems by Mead and Conway*, chapter 8.3. Addison-Wesley, 1980.

[Loveman 77] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the Association for Computing Machinery*, 24(1):121–145, January 1977.

[Reddy 88] U. S. Reddy. Program transformation without folding (technical summary). Technical report, University of Illinois at Urbana-Champaign, January 1988.