Algorithm Design and Parallel Program Development through Formal Specifications

David L. Zimmerman  
Kestrel Institute

Michael R. Lowry  
Kestrel Institute

ABSTRACT

This paper describes two components of the Kestrel Interactive Development system (KIDS). The overall methodology is top-down refinement of formal specifications; software design decisions are factored into logical steps along an extended “what-to-how” spectrum.

The algorithm design component is based upon reusing the structure common to a class of algorithms, such as local search. By formalizing this structure as an abstract theory, the system can then instantiate its parameters to domain specific functions, to obtain a high-level program specification. The parallel program development component supports a generalized data flow model, in which a computation is broken down into a collection of distinct atomic units. These are then scheduled and assigned to individual processors through formalisms expressing resource allocation constraints. We first illustrate algorithm design with the development of the simplex algorithm from a high-level specification, and then develop a parallel version of one sub-operation: the dot product computation.

1 Introduction

Basing the software development process on formal specifications offers significant improvements for software design, validation, maintenance, and reuse. Since formal specifications have a well defined semantics, knowledge-based tools which use correctness-preserving transformations can support their refinement to efficient machine-level code. The capability of compiling and executing high-level specifications enables testing and validation early in the software lifecycle. High-level specifications are easier to maintain, modify, and reuse than source code since they lack the implementation details.

A given program implementation can be viewed as a combination of several forms of knowledge, such as problem domain knowledge, algorithm design knowledge, and target machine knowledge. To automate software reuse, these various sources of knowledge should be machine-encoded and factored into logically orthogonal components, rather than being embedded in the source code. Research at Kestrel Institute has focused on tools for developing software through several levels of formal specifications from which efficient code can then be synthesized. This approach raises the software reuse problem to that of revising the specifications at appropriate levels of program refinement and automated reuse of the associated knowledge base components.

Software development environments for designing correct and efficient code from formal specifications have been a goal of Artificial Intelligence and related fields for twenty years [6]. There are several difficult research problems which are being solved in order to make this goal a practical reality. The first is formal languages for representing specifications and programs at various levels of abstraction [15]; most formal languages are augmentations of mathematical logic. The second is to encode software design knowledge in machine-useable form so that it can be reused and automatically applied. This knowledge is usually encoded as either transformation rules or inference rules [13]. The third problem is to produce knowledge-based assistants which automatically or interactively synthesize source code from formal specifications. This requires the knowledge to be factored into orthogonal components which can be separately applied, yet when taken together, have a wide range of coverage. Factoring the knowledge is necessary to avoid combinatorial explosion, where a computer effectively grinds to a halt because of the large number of possibilities at each step.

In contrast to work such as Manna and Waldinger [14] and Constable [4] which is primarily concerned with providing a rigorous account of the mathematical basis of programming language constructs, the research on KIDS is primarily focused on providing an environment for software development. For example, in the NuPrl system [4] a user guides NuPrl through a long and detailed proof in order to develop a program. In KIDS the user makes high-level decisions and leaves the details of theorem proving to the system. In contrast to domain specific approaches such as advocated by Barstow [3], KIDS is a general purpose system. It can be applied to any domain as long as it is provided with a suitable domain theory.

This paper describes two components of our knowledge-based program development system. Section two presents a brief overview of KIDS [17], and the REFINE™ [1] programming environment upon which it is built. Section three describes Strata—an algorithm design system based on algorithm theories, which formalize the structure of a class of algorithms. Section four describes Kappa—a prototype system for transforming high-level specifications without explicitly parallel control structures into concurrent implementations.
on a range of parallel architectures. Section five offers a brief summary.

2 An Overview of KIDS

KIDS provides an open architecture for experimenting with various transformational software development components. It is used to interactively develop formal specifications into correct and efficient programs. Dozens of programs have been derived using the KIDS environment and we believe that it is close to the point where it can be used for routine programming.

The KIDS environment provides languages for expressing various forms of knowledge, and tools for utilizing this knowledge in software development. It includes components for performing deductive inference, algorithm design, expression simplification, finite differencing, a form of partial evaluation, and developing code for parallel architectures. These components raise the conceptual level from which the programmer can obtain correct and efficient executable code.

2.1 The User Interface

The user of KIDS develops a given formal specification into code by interactively applying a sequence of high-level transformations. At any intermediate point during the development, the user views a partially implemented specification annotated with input assumptions, invariants, and output conditions. A mouse is used to select a transformation from a command menu and to select a subexpression of the specification to which the command is applied. In effect, the user makes high-level design decisions and the system carries them out.

2.2 Underlying technology

KIDS is built on top of REFINE, a commercial knowledge-based programming environment. REFINE provides (i) an object-oriented data base that is used to represent software related objects via annotated abstract syntax trees, (ii) grammar-based parser/unparsers that translate between text and internal representations, (iii) a very-high-level language including transformation and pattern constructs that support the creation of rules, and (iv) a compiler which produces COMMONLISP code from a subset of the language. The entire KIDS system is written in REFINE and all of its operations work on the annotated abstract syntax tree representations in the REFINE object base.

KIDS uses an extension of theorem proving technology called deductive inference. Deductive inference is necessary whenever it is desired to correctly apply general knowledge to particular problems. RAINBOW is a general-purpose deductive inference system, which transforms a formula into another formula bearing a specified relationship to the first [16]. As special cases it can perform first-order theorem-proving and formula simplification. It also allows the inference of sufficient conditions or necessary conditions of a formula.

This flexibility allows us to formulate a variety of design and optimization problems as inference tasks.

3 The Strata Subsystem

Strata is an algorithm design system based on algorithm theories which represent the structure common to a class of algorithms, such as local search [10]. Algorithm theories abstract away implementation concerns about control strategy, target programming language, and target architecture [18]. In the KIDS methodology, these concerns are factored in as later decisions in the design process.

Associated with each algorithm theory is a design tactic which semi-automatically constructs an algorithm given a formal problem specification. We will illustrate our approach to algorithm design with Strata's derivation of the simplex algorithm from the specification of linear optimization problems. In this derivation Strata uses the design tactic for local search algorithms.

A design tactic combines several sources of machine usable knowledge in order to derive a high-level algorithm specification from the formal specification of a problem. The basic means of combining this knowledge is directed inference. The first source of knowledge is the formal problem specification itself. The second source of knowledge is a domain theory which defines the semantics of the objects, functions, and predicates of the problem specification, as well as their semantics of basic operations of the domain. For the linear optimization problem this domain is linear algebra; the domain theory was encoded as axioms in RAINBOW. The third source of knowledge used by a design tactic is a collection of library theories, which are specializations of part of the algorithm theory underlying the design tactic. The library theories for local search are parameterized by the type of feasible solution. The final source of knowledge used by a design tactic is the algorithm theory itself. The design tactic uses the algorithm theory to extend and specialize a library theory in order to derive an algorithm for a problem specification.

The basic method followed by a design tactic consists of the following steps:

1. The tactic uses the input and output type specification to retrieve candidate library theories.
2. The tactic uses directed inference to determine which candidate library theories satisfy necessary conditions stated in the problem specification. If successful, the parameters of the library theory are instantiated to the problem.
3. The tactic then uses directed inference to further extend and specialize the instantiated library theory, according to the axioms of the algorithm theory.
4. The design tactic constructs a high-level specification for an algorithm from this extended and instantiated theory.
3.1 Linear Optimization

An optimization problem is formally specified by two components: a basic problem structure and a cost function. The basic problem structure determines the feasible solutions for any given input, while the cost function establishes a preference ordering on feasible solutions. For example, the specification for linear optimization that Strata begins with is shown below:

```
(defobject linear-optimization
  function(A : RMat, b : Rvec, c : Rvec)
  :set(column) = some(y)
  (extremal(y, λ(y1, y2))
   (dotp(c, solve(y1, b)) < dotp(c, solve(y2, b)));
   { basis | (basis ⊆ columns(A))
     & (size(basis) = size(row(A))
     & non-neg-coordinates(solve(basis, b)) })
```

The first two lines specify that the input consists of a matrix \( A \), a vector \( b \) and a cost vector \( c \). The type of feasible solutions is subsets of the columns in \( A \). These subsets are a representation for co-ordinate vectors which are obtained through the function \( \text{solve}(y, b) \). The cost function is the dot product of such a vector with \( c \). The last three lines specify the invariant properties of feasible solutions: the number of columns must equal the number of rows in matrix \( A \), and the coordinate vector of a feasible solution is non-negative. Strata's internal representation of this specification is an abstract syntax tree where the input variables, output variables, and invariant properties of feasible solutions are all made explicit.

3.2 Local Search Algorithms

Local search algorithms search a large space by successive searches over small neighborhoods. A local search algorithm is derived by extending an optimization problem with an instantiation of the structure common to all local search algorithms. This structure is formalized as the algorithm theory of local search. The parameters include a basic iterator and filters which specialize the basic iterator for feasibility and hill-climbing. Figure 1 illustrates the abstract structure of local search algorithms.

The basic iterator defines an enlarged neighborhood of each solution: the adjacent basic solutions. The basic iterator is defined as a relation between a current solution, neighboring solutions, and exchange variables. The exchange variables denote components of the current solution which are exchanged to yield neighboring solutions. Intersecting this neighborhood with the feasible solutions yields the adjacent feasible solutions. This intersection is defined by specializing the basic iterator.

More specifically, the design tactic uses directed inference to deduce necessary conditions on the exchange variables. Finally, the iterator is again specialized to only derive adjacent feasible solutions which improve the cost of the current solution. Once again, the design tactic uses directed inference to deduce necessary conditions on the exchange variables.

![Figure 1: Structure of Local Search Algorithms](image)

This design tactic can be automated because it factors a difficult inference step into simpler steps: an extension step and subsequent specialization steps. In effect, the design tactic for local search algorithms is itself a local search algorithm. Starting with a relaxed set of invariant properties, Strata then "hill-climbs" by specializing. Strata hill-climbs in the space of iterator definitions.

3.3 Deriving the Simplex Algorithm

Strata derives the basic iterator by retrieving candidate basic transformations and their invariants from a library indexed by the parametrized output type of feasible solutions. For example, the type of feasible solutions for the linear optimization problem is subsets of columns. Thus the parametrized output type is subsets of \( \text{any-type} \), where \( \text{any-type} \) is a parameter which can be instantiated to any set. The library is based on the mathematics of group theory and was obtained by reversing the rules of invariant logic [12]; the theoretical basis is explained in detail in other papers [11]. One of the candidate library theories retrieved for the simplex algorithm is shown below:

![Diagram of Simplex Algorithm](image)
Local-Search-Theory is-subsets

Input-vars finite-set: set(any-type),
N: integer

Output-vars old-subset: set(any-type)

Transform-vars new-subset: set(any-type)
x-vars old-elem: any-type,
new-elem: any-type

Transform-definition
new-subset ⊆ finite-set
& size(new-subset) = N
& new-elem ∈ finite-set
& new-elem ∉ old-subset
& old-elem ∈ old-subset
& new-subset =
( old-subset U { new-elem }) - { old-elem }

Invariant old-subset ⊆ finite-set
& size(old-subset) = N

For each candidate library theory, the design tactic sets up the following specification for directed inference:

∀ input, output, library-vars
Input-condition(input)
& Feasible(input, output)
⇒ Invariant(library-vars)

The successful verification of this theorem instantiates the parameters of the library theory. In other words, the library theory is tailored to the particular problem specification. For the simplex example, subsets of any-type is instantiated to subsets of columns of the matrix A, and the size of the subsets is instantiated to the number of rows in the matrix A.

The instantiated transform-definition is the specification of the basic iterator. This definition specifies how a new element is swapped for an old element in order to generate neighboring solutions.

In the simplex algorithm derivation, Strata specializes the basic iterator to a feasible iterator by restricting the exchanges to ones which also keep invariant the non-negativity of the coordinate vector. The feasible iterator is further specialized to a hill-climbing iterator by restricting the exchanges to ones which improve the cost function.

Strata derives the feasible iterator by instantiating the feasibility-constraint parameter of the local search algorithm theory. Directed inference derives necessary conditions of feasibility on the variables new-elem and old-elem. This moves constraints on feasibility into the iterator of feasible solutions. If equivalent conditions are derived, then the iterator only generates feasible solutions; the feasibility test then becomes redundant and can be eliminated. The abstract inference specification shown below for the feasibility-constraint parameter is taken directly from the theory of local search. The instantiation of the inference specification for local optimization is shown, and the definition returned by directed inference follows.

The axiom in the abstract theory of local search:

∀ input, current-soln, new-soln, x-vars
Input-condition(input)
& Feasible(input, current-soln)
& Feasible(input, new-soln)
⇒ Feasibility-constraint(current-soln, new-soln, x-vars)

The abstract axiom as instantiated for linear optimization:

∀ A, b, c, current-basis, new-basis, old,
non-neg-coordinates(solve(current-basis, b))
& size(current-basis) = size(Rows(A))
& non-neg-coordinates(solve(new-basis, b))
& size(new-basis) = size(Rows(A))
& new-basis = insert(new, delete(old, current-basis))
⇒ Feasibility-constraint(current-basis, new, old)

The definition returned by directed inference using the domain theory of linear algebra:

Feasibility-constraint(current-basis, new, old) =
solve(current-basis, A[new])[old] <
solve(current-basis, b)[old]

Recall that solve(y, b) returns a vector, so the two sides of the inequality are scalars obtained by indexing the vectors with the variable old. This feasibility constraint is formulated without explicit reference to the variable new-basis, which can therefore be eliminated.

Strata’s derivation of the hill-climbing iterator parallels that of the feasible iterator; it is defined by instantiating the improvement-constraint parameter. In this case, the definition returned by directed inference using domain knowledge of linear algebra is:

Improvement-constraint(current-soln, new, old) =
dotp(c, solve(current-basis, A[new])) < c[new]

After Strata specializes the basic iterator by incorporating the constraints on feasibility and hill climbing, it determines necessary and sufficient conditions for a local optimum to be a global optimum. For the simplex algorithm derived above.
a sufficient condition is “true”, which means that when the algorithm attains a local optimum it has found a global optimum.

In the final step of the design tactic, *Strata* assembles three programs from the instantiated theory. The first is an *INITIAL-VALUE* program which produces an arbitrary feasible output given an input. The second is a linear recursive hill-climber which iterates from feasible solution to better feasible solution until a local optimum is reached. The third program is a shell program which calls the recursive hill-climber with the output of the *INITIAL-VALUE* program.

To summarize, the algorithm design component of KIDS semi-automatically develops a high-level specification for an algorithm from the formal specification of a problem by combining several sources of knowledge. This algorithm specification can then be further transformed through successive stages by factoring in other types of knowledge, such as general algorithm optimization knowledge and target machine knowledge. The next section will describe ongoing work at Kestrel for developing parallel programs from high level specifications of algorithms.

4 The *Kappa* Subsystem

*Kappa* stands for Kestrel’s Automated Parallel Programming Assistant, to be developed and implemented as a series of increasingly powerful prototypes [8]. It provides the framework for efficiently implementing high level specifications on commercially available, general purpose multiprocessors. *Kappa* progressively transforms a formal problem specification into a sequence of increasingly lower level descriptions. Several lower level representation forms are introduced in the path between an abstract specification and production of native code for a particular machine.

*Kappa* accepts an initial problem specification written in a declarative style, typically given as a mutually recursive collection of object definitions. Such specifications can be derived within KIDS by applying various algorithm design tactics, as illustrated in the previous section. *Kappa* supports a data flow model [2], in which any object of the computation may be assigned a value only once. In the data flow model, the primitive operations of the problem, together with their interdependencies, are represented as a graph [5]. The graph representation of a procedure has the virtue that it does not constrain parallelism beyond the requirements imposed by direct data dependencies among the operations.

When specialized data flow architectures are used, no attempt is made to group the primitive operations expressed within the graph, which tend to be fairly simple. However, compiling data flow specifications for other than data flow architectures requires some means for partitioning the operations into individual processes, and scheduling the processes for execution on individual processors. Partitioning and scheduling should be handled so as to minimize communication and retention of data, yet make maximum use of available resources.

Our methodology is based upon controlling resource allocation across a collection of potential obligations to produce required values. We first describe the structure of the entire computation, dividing it into units that we will not consider further dividing. These units, termed responsibilities, are just the pairing of some (abstract) map with a single element of its domain. Each responsibility has just one role: to compute the map’s value at that domain point. Whether it ever does so depends on its acquiring sufficient resources to perform the computation. Responsibilities thus provide a name space which we use as the basis for partitioning and scheduling, in terms of resource allocation.

This scheme provides a great deal of flexibility, allowing us to generalize along two axes: an eagerness/laziness axis and a compile-time/run-time decision axis. For instance, a more eager resource allocation strategy will provide resources to a responsibility earlier, when the evidence is less complete that its results will eventually be needed. As another example, making allocation decisions during compilation suggests providing resources to groups of responsibilities that have been determined, at compile time, to have a correlated demand. In contrast to its value (which can only become defined), a given responsibility’s entitlement to resources can change during the course of the computation. This permits *Kappa* to express a variety of resource allocation strategies.

The following subsections discuss the representation languages of *Kappa* and describe important parts of the transformation process. The unifying representation for this process is an intermediate form which supports explicit naming of responsibilities and logical constructs for managing the assignment of resources to them. As example, we illustrate the transformation to intermediate form of a formal specification for dot product, one of the subcomputations of the simplex algorithm developed above.

4.1 Source Language

A functional subset of *REFINE™* constitutes the problem *Source Language*. The problem is formally specified as a set of recursion equations, using constructs from first-order logic. The language contains no explicitly parallel constructs; instead, the system automatically derives and makes progressively more explicit how program concurrency will be organized. This approach avoids the inclusion of irrelevant sequencing details at higher levels of program description.

The primitive types of the language are the usual scalar types, such as integers, reals, booleans, and characters. Its composite type constructors include the usual type constructors of high level programming languages, as well as set-theoretic types, such as sets and maps, provided by *REFINE*. A map may be specified both literally, by pointwise assignment, or algorithmically, as a set of constraints on its range values. The source language includes higher order operators such as reduction, composition, and closure.

We can formally specify the standard definition of “dot product” as follows:
\[\text{dotp} : \text{map}(\text{seq}(\text{integer}) \times \text{seq}(\text{integer})) \rightarrow \text{real}\]

\[
\forall v_1, v_2 : \text{seq}(\text{integer}), n : \text{integer}
\text{dotp}(v_1, v_2, n) = \red\left(\sum_1^n [v_1(i) \times v_2(k)] \mid (i) 1 \leq i \leq n\right)
\]

This specification can be read as “Define \text{dotp} such that for all \(v_1, v_2,\) and \(n\).” It uses the higher-order operator \text{reduce}, which takes a commutative, associative, binary operator with identity (here \(+\)) and a “bag” or multiset. Here it specifies the summing of paired vector element products over all the index points from 1 to \(n\).

### 4.2 Normalized Form

As its first transformation step, \text{Kappa} decomposes the composite type constructions in the original problem specification into standard representation forms over a more limited set of constructors. The normalized form is expressed within a strict subset of the full source specification language; it admits only maps and tuples as type constructors. Component types in the normalized form are restricted to scalars and scalar tuples. Since the size of a tuple can be determined statically, from its type specification, these restrictions ensure that every data object within the normalized specification has a known, fixed size.

Fixing the size of a problem’s data objects allows later transformation steps to make knowledgeable decisions about tradeoffs in organizing the computation. In addition to the usual time and space tradeoffs in sequential programming, communication costs must also be considered. Point-to-point communication latencies depend only on the routing circuitry between processing elements and the size of the data object to be transmitted. Thus, with maximum object sizes established, the communication delay for each potential data transmission can be bounded by a constant proportional to the inherent delay of the communication paths to be used.

The normalization process comprises two basic transformation steps: representation of composite objects as maps, and recomposition of maps so that neither their domains nor ranges are composite objects. First, we represent sets, sequences, and bags as semi-characteristic functions (maps). So if \(S : \text{set}(x)\), we create a representation function \(S' : x \rightarrow \text{boolean}\), and arrange that \(S'(x) = \text{true}\) just when \(x \in S\). Next, we encode each map value used in the domain of another map as a new blank atom (a symbol with no semantics except equality) together with a representation map to correlate its domain/range pairs.

For example, to normalize

\[M(N)\text{ where } M : [\tau_1 \rightarrow \tau_2] \rightarrow \tau_3\]

we create

\[M' : \text{atom} \rightarrow \tau_3\text{ and } R : [\text{atom} \times \tau_3] \rightarrow \tau_3\]

then replace \(M(N)\) with \(M'(\alpha_N)\), where \(\alpha_N\) is an atom specific to a particular value of \(N\), and establish that

\[\forall v \in \tau_1 R(\alpha_N, v) = N(v)\]

When the range of a map is itself a map, we first “deCurry,” transforming

\[M(u)(v)\text{ where } M : \tau_3 \rightarrow [\tau_1 \rightarrow \tau_2]\]

into

\[M'(u, v)\text{ with } M' : [\tau_3 \times \tau_1] \rightarrow \tau_2\]

Where the specification has \(M(\ldots) = \alpha_N\), we’ll put \(M(\ldots) = \alpha_N\), and construct a representation function similar to the above, such that

\[v \in \tau_1 \implies R(\alpha_N, v) = N(v)\]

In both cases we can then replace all references to \(n(v)\) with the corresponding reference to \(R(\alpha_N, v)\).

Applying these normalizations, along with algorithm refinement of the \text{reduce} operator, to the original specification we obtain:

\[\text{dotp} : \text{map}(\text{atom} \times \text{atom} \times \text{integer} \rightarrow \text{real}),\]

\[S' : \text{map}(\text{atom} \times \text{integer} \rightarrow \text{real}),\]

\[\text{Rgbag} : \text{map}(\text{atom} \times \text{integer} \rightarrow \text{real}),\]

\[
\text{in } \forall a_1 : \text{atom}, a_2 : \text{atom}, n : \text{integer}
\text{dotp}(a_1, a_2, n) = \red(\text{red}(a_R, 1, n)) \quad ; \text{call 1}
\& \forall (i : \text{integer}) 1 \leq i \leq n \implies \red\left(\text{Rgbag}(a_R, i) = S'(a_1, i) \times S'(a_2, i)\right)
\]

\[\text{red} : \text{map}(\text{atom} \times \text{atom} \times \text{integer} \rightarrow \text{real})\]

\[
\text{in } \forall a_3 : \text{atom}, b : \text{integer}, t : \text{integer}
\text{red}(a_3, b, t) = \text{red}(a_3, b, t)
\& b \neq t \implies \text{red}(a_3, b, t) = \red(\text{red}(a_3, b, \lceil(b + t)/2\rceil)\)
\]

The original \text{dotp} has been transformed to take two atoms and an integer, instead of two vectors; the representation map \(S\) provides the correlation of index elements to vector values. The original \text{reduce} operator has been expanded using a divide-and-conquer strategy, which produces a binary tree of partial inner products. The new map \text{Rgbag} supplies the products at individual index points through the atom \(a_R\), and \text{red} recursively sums them. Its parameters \(b\) and \(t\) denote the bottom and top indices for the current partial sum; “call 1” is just \text{dotp}, the sum across all index points. The recursion bottoms out when \(b\) and \(t\) are equal. Otherwise, the current index range is split into equal subranges at “call 2” and “call 3.”
4.3 Intermediate Form

Most of the work in Kappa revolves around the intermediate form, also known as the Intermediate Language for Kappa (ILK). It is the central, unifying representation within the transformation process. We also envision its use as a potential entry point for specifications generated by other systems. Within Kappa, successive passes over the current intermediate form description apply various rule-based transformations, and progressively augment it with additional detail in the form of further specifications.

The intermediate form abstracts all computational effort as distinct responsibilities: individual applications of some map to an element of its domain. Each responsibility denotes a value, defined by the specified map and domain element pair, which may not be redefined during the computer's lifetime. The abstraction includes both the computation necessary to develop the value, and the value's subsequent retention, so that it can be made available to other responsibilities as needed. The notation \( (f, a_1, \ldots, a_n) \) is read "f at \( a_1, \ldots, a_n \)" and denotes application of the map \( f \) to the tuple \( \langle a_1, \ldots, a_n \rangle \). For an ordinary value name, \( x \), the corresponding responsibility \( \{x\} \) is just the determination and retention of the single range value named by \( x \).

An abstract problem description generally contains a collection of mutually recursive definitions of mappings and values. These definitions often specify formal relations which must hold between the values of the various maps at different points of their respective domains. The domain points of interest will often be related through existential quantifiers internal to a map’s definition. These “local variables” could potentially be constrained to take on distinct values, through functional dependence on the map’s parameters, and thereby violate the “single value per responsibility” restriction. In addition, it might be desirable to perform a computation independently in two places, rather than find and transmit a previous result through the multiprocessor’s communication network.

Kappa addresses these issues through two related transformations: environment explication and quantifier restructuring. The basic idea is to create an explicit object, an “environment” to represent the evaluation context of each map within the definition, thus exposing any dependencies on map parameters. That is, we transform a map definition \( M \), with formal parameters \( P \), and local variables \( L \), to a new definition \( M' \), with a single parameter \( e \). For each value name \( v \) in \( L \cup P \), a new \( \exists \)-quantified map \( v' \) is defined, from \("M-environments" to a value of the appropriate type. Then we replace each use of a value name \( v \) in \( L \cup P \) with the corresponding reference to \( v'(e) \).

Finally, any conjunct in the specification that includes a reference to \( M \), say \( Q(M(a, b, \ldots)) \), is transformed to

\[ \exists e \ (Q(M'(e)) \land p_1'(e) = a \land p_2'(e) = b \& \ldots) \]

where \( p_1, p_2, \ldots \) are the formal parameters of \( M \). The potential for multiple values being bound to a single name is not limited to local variables of map definitions; it can occur anywhere an existential quantifier is enclosed within a universal quantifier. Therefore, we perform quantifier restructuring, expanding the scopes of all existential quantifiers to enclose universal ones. More formally, where \( \tau \) are types, we transform

\[ \forall x : \tau_1 \exists y : \tau_2 \ldots \]

into

\[ \exists y' : [\tau_1 \rightarrow \tau_2] \forall x : \tau_1 \ldots \]

throughout the specification. Note that in map definitions, by simply introducing a universally quantified environment variable and replacing the parameter references with map references as shown above, we would expose any local variables’ dependence on the environment. Quantifier restructuring would then complete the work of environment explication, in addition to its normalizing role.

Additional steps orient the equations, determining which way the equalities in the specification will be “driven,” and introduce new maps to relate function “callers” to “ callees.” These additions are shown below through the use of \( :=' \) for the oriented equations and the comments to the right.

We have followed the notational examples above in choosing names for the environment explication maps, thus \( dotp \) for \( dotp \), \( n' \) for \( n \), and so forth.

\begin{align*}
\text{dotp} &: \text{map}(\text{env} \rightarrow \text{integer}), \\
S' &: \text{map}(\text{atom} \times \text{integer} \rightarrow \text{real}), \\
\text{Rbag} &: \text{map}(\text{atom} \times \text{integer} \rightarrow \text{real}), \\
\text{red} &: \text{map}(\text{env} \rightarrow \text{integer}), \\
\text{map for call 1} \\
\text{map for call 2} \\
\text{map for call 3}
\end{align*}

\[ \forall e, i (\text{dotp}'(e) := \text{red}'(m_{C1}(e))) \]

\[ \& (a_0'(m_{C1}(e)), \theta'(m_{C1}(e)), t'(m_{C1}(e))) = \\
(\theta'(e), 1, n'(e)) \]

\[ \& i \leq i \leq n'(e) \implies \text{Rbag}(a_0(e), i) := \\
S'(a_0'(e), i) \ast S'(a_0'(e), i) \]

\[ \& \forall e, \theta(e) = \theta'(e) \implies \text{red}'(e) := \\
\text{Rbag} + \theta(e) \]

\[ \& \theta(e) \neq \theta'(e) \implies \text{red}'(e) := \\
\text{red}'(m_{C2}(e)) + \text{red}'(m_{C3}(e)) \]

\[ \& (a_0'(m_{C2}(e)), \theta'(m_{C2}(e)), t'(m_{C2}(e))) = \\
(\theta'(e), 1, \xi'(e) + t'(e)/2) \]

\[ \& (a_0'(m_{C3}(e)), \theta'(m_{C3}(e)), t'(m_{C3}(e))) = \\
(\theta'(e), [\theta'(e) + t'(e)/2], t'(e)) \]
The oriented equations between map applications enforce the problem's data dependencies, and indicate where environment mappings must be supplied. The environment mappings, together with the tuple equalities specified, explicate parameter transmission. Now all that remains is the assignment of responsibilities to processors and introduction of resource allocation constraints.

4.4 Resource Allocation

To specify processor assignment and resource allocation, we augment the intermediate form description with several varieties of metapredicates. For example, process $p$-dotp($e$) has

$$\langle (a_1, \Theta e) \rangle, \langle (a_2, \Theta e) \rangle, \langle (n, \Theta e) \rangle, \langle (a_8, \Theta e) \rangle$$

.. collects responsibilities from one environment, and declares them to share a process. The form $\langle (f, \Theta z_1, \Theta z_2, \ldots, \Theta z_m) \rangle$ denotes a prototypical responsibility scheme, which is bound to a specific responsibility exactly when $z_1$ and $z_m$ have values. Any combination of the responsibility arguments may be scheme variables instead of fixed values. Quantifiers attached to the process . . . has declaration permit indefinite groups of responsibilities to share a process.

To describe and manage resource allocation to responsibilities we consider a given responsibility to be in one of four states:

- unrealized not entitled to any resources
- realized entitled to resources for computation
- known having had a value computed
- obsolete having a value no longer needed

Metapredicates describing these states differ from ordinary predicates in that (i) their objects are responsibilities, rather than objects of the problem specification, and (ii) their values change during a computation, rather than merely becoming defined. For instance,

$$\text{known } (a', \Theta e) \Rightarrow \text{realized } (m_{C1}, \Theta e)$$

$$\text{known } (m_{C1}, \Theta e) \Rightarrow \text{realized } (a_1, \Theta m_{C1}(e)),$$

expresses that resources will be allocated to constructing the environment for ref' once the first argument vector to dotp has been given a value, and that resources for producing the map's parameters will be allocated when the evaluation of that environment completes.

As an aid in developing the forms above, Kappa will draw upon instance advice supplied by the programmer and/or extracted from the KIDS knowledge base. Such information will state, for example, the likely size of the maps to be constructed for a specific problem instance, or the relative overhead of process creation and process communication on a particular target architecture. Initially, these characterizations will be supplied only by knowledgeable developers, in terms of objects defined within the intermediate form. In later implementation stages, we imagine that the system will be able to interactively query users, in terms of objects defined within the source specification.

4.5 Code Generation

A fairly wide range of multi-processor systems have already been constructed, and many more potential architectures have been proposed. The future will bring even more designs and further system realizations. To address this diversity, Kappa will translate the intermediate form description into one of several possible retargetable forms. The intent here is to extract the common features of a family of parallel architectures.

We envision that the wide variety of available architectures will be partitionable into a relatively small number of basic classes. Each particular class will be paired with a corresponding retargetable language form. The partitioning is not meant to capture all the details of different architectures, only their general characterization, in terms of basic computational model, individual processor capabilities, and communication primitives.

Target architecture advice figures most prominently here, both in specialization to the machine class, and through any additional information available to further distinguish the intended member. Early versions of Kappa will only recognize a small subset of identified classes, and make limited use of such additional information. A single class will be represented in the current development version, and any distinguishing information offered for a particular target will simply be stored within the knowledge base. Incremental growth of this architectural knowledge base will make later versions capable of more sophisticated implementation choices, and fuller exploitation of specific machines' features.

The retargetable forms constitute the most concrete class of "specification" languages used within Kappa. They make concurrency explicit by arranging for the creation and destruction of processes, assignment to processors where appropriate, and direct expression of data communication as required. The concrete scheduling and synchronization constraints represented in these forms are derived from the more abstract description of resource allocation dependencies discussed above.

As the last step in the synthesis, native code for a particular machine is generated directly from the retargetable form. The native code implementation will typically be expressed within a superset of some language such as COMMON LISP or C, whatever the target machine's manufacturer supports as its principal programming language. Such languages may be more or less standard, depending on the possible inclusion of peculiar parallel operation forms, distinct message passing constructs, and so forth. Nonetheless, we expect that, within a given class, the retargetable representation(s) will differ very little, leaving only the syntactic transformations necessary to synthesize native code for the specific target machine.
5 Summary

The construction of a computer program, like any complex artifact, is based on several kinds of knowledge. Knowledge about the particular problem being solved, knowledge about the application domain, knowledge about algorithms, data structures, and optimization techniques, and knowledge about target machine architectures are all factored into the design. We have examined two components of a knowledge-based program development and transformation system under construction at Kestrel Institute. Both components follow the same basic methodology: incremental refinement of formal specifications through interaction with automated transformation tools.

An essential aspect of this methodology is its grounding in formal theories from various domains of interest. Formal theories both abstract away irrelevant details, and provide the only likely basis for verifying that program implementations accurately reflect their design specifications. By factoring such theories among distinct, but interrelated tools, we construct a multi-level description of a program, which enforces a wide range of internal consistency constraints.

Thus, we may respecify a new set of design and implementation constraints at any of the description domains, and rederive the other components (almost) automatically. The extension and transfer of such capabilities, out of research about target machine architectures are all factored into the design. Both components follow the same basic methodology: incremental refinement of formal specifications through interaction with automated transformation tools.

References