A Prototyping Language for Rapid Reuse

Volume I: Technical Proposal

Prepared for
Defense Advanced Research Projects Agency
Attn: Dr. Jack T. Schwartz
Information Science and Technology Office
1400 Wilson Blvd., 7th Floor
Arlington, VA 22209-2308

In Response to Broad Agency Announcement 8908

Prepared by

Rensselaer Polytechnic Institute
Troy, New York

Hewlett–Packard Laboratories
Palo Alto, California

State University of New York at Albany
Albany, New York

Contents

1 Introduction

Rensselaer Polytechnic Institute, Hewlett–Packard Laboratories, and The State University of New York at Albany jointly propose to develop an Initial Design for a new language, Æ,[1] for rapid construction of software prototypes of systems to be ultimately delivered in Ada. The key idea of our approach is to make the construction and use of *generic libraries of software components* the cornerstone of programming methodology—at both the prototyping and the production system development levels. By doing so, we expect to dramatically improve the productivity of software development and the quality (including both reliability and maintainability) of the delivered product.

We have already made substantial progress in generic library construction for use in production system development in Ada [22,23,24]. But Ada, like other production programming languages, lacks key features required for rapid prototyping. On the other hand, prototypes developed in languages such as Common Lisp or Smalltalk cannot take advantage of Ada's strengths such as the extensive static checking allowed by strong–typing and other features supporting the modular organization of large systems; nor does the model of computation supported by these languages map well into Ada.

We therefore conclude that the best approach is to base the design of Æ on Ada itself, by extending Ada with a small number of constructs carefully chosen to provide the major advantages for prototyping found in languages such as Common Lisp or Smalltalk, but sufficiently limited that prototypes using these constructs can still be easily refined into Ada. This extended language will allow prototypes to be organized with even greater modularity and more rigorous static–checking than Ada itself supports. In particular, Æ will include many prototyping–level generic libraries; corresponding production–level generic library components in Ada; and straightforward paths of refinement of prototypes into production code, based on extensive use of the libraries at each level.

Generic libraries Since generic libraries play such a central role in our approach, it is important to make clear how our notion of generic libraries, as developed in [22,23], provides much greater support for reusability than older notions of software libraries. An obstacle that has often prevented widespread use of software libraries is lack of flexibility of the components. Too often one will find a component that is almost what is needed, but not quite; eventually one abandons the library and reverts to programming every operation from scratch. We believe that we now have the technology for library construction in place to overcome this problem, by making software components highly

[1] Æ is pronounced as a long E. It can also be written AE.

generic: we abstract from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

The mechanisms provided by Ada generic units already make it possible to structure libraries according to this approach and still obtain highly efficient production code, since the compiler has the opportunity to consider the special optimizations that are enabled when actual parameters are instantiated into generics (unlike other mechanisms in which instantiation does not occur until run–time). Æ will include language extensions closely related to Ada generics, to broaden and simplify their use during prototyping.

Behavioral and structural prototyping The proposed language extensions in Æ divide into two classes: one supporting *behavioral prototyping* (*what* the system being prototyped is supposed to do, a black–box model), and the other aimed primarily at supporting *structural prototyping* (*how* the black–box behavior will be achieved) [6].

The constructs for structural prototyping provide, relative to Ada, better support for *organizing generic libraries*; for *higher–order programming*; for *object–oriented programming*; and for *parallel and distributed computing.* For all these constructs, mappings will be defined to transform prototypes into equivalent Ada code.

For behavioral prototyping, we introduce constructs that associate a *meaning* with each of the major abstraction mechanisms of the language (including those of Ada), expressed primarily with *conditional equations.*[2] These constructs can be used in several important ways to specify behavior, based both on the ability to do extensive semantic analysis and to execute the constructs directly by treating the equations as *rewrite rules.* Meanings can also aid in detecting data dependencies that may affect which parts of a program can be executed in parallel and in automatically imposing task structure on packages and their Æ generalization, forms (see Section 2.1.5).

Real–time and AI concerns Two areas are of special concern for production of many kinds of defense related software systems: real–time constraints on embedded systems, particularly distributed systems; and use of artificial intelligence related methods. We address these areas in two ways: first, by our approach of stressing the role of libraries and reusability of code; and second, by providing assistance, in both Æ and its environment, for configuring *specialized run–time systems.*

[2] E.g., for an Ada package specification, **package** *Name* **is** ..., a corresponding meaning can be declared with the construct **package** *Name* **meaning is** .... This meaning declaration would contain *descriptors* about the semantics of functions and procedures of the package and *axioms* expressed as conditional equations.

In the real–time area, as Firth [7] has pointed out, reusable components can have an important positive impact on performance prediction, through design of systems "so that they can use existing code that has already been measured .... If we can build a real time system largely out of tested and assessed components, with simple interfaces of predictable behavior, we can estimate performance of the whole by synthesizing the known performance of the components." In the AI area as well, it will be possible to develop extensive libraries, for both the Æ prototyping level and the Ada production level, of flexible data structures, pattern-matching algorithms, and algorithmic and heuristic search methods, including rule-based inference methods.

Ada itself has been criticized for not providing adequate support for real–time systems (e.g., see [2]), and development of such support at the language–level is still an open research area. A pragmatic approach for the short–term is to support the construction of special run–time systems, tailored to specific application areas. Such run–time systems for Ada can be built from highly modular components that can be configured to provide just the particular functionality needed, with minimal code size (e.g., see [1]). There are, however, many tradeoffs that must be carefully managed. At the language level, we propose to develop a set of pragmas that control the structure of run–time systems, in terms of memory–management, single or multiple address spaces, context–switching mechanisms, etc.

Specialized run–time systems are also a key to enabling AI techniques to be used in embedded systems, since they can be configured to supply minimal run-time support, for example, for the particular inference methods used in an application.

Common Prototyping Environment Though concerned mainly with production–system issues, specialized run–time system configuration requires insights about the application area that are best obtained from prototyping–level activity. For this reason, in considering the design of a Common Prototyping Environment, we propose that a prominent role be given to *measurement tools* for experimenting with application–specific run–time systems in order to meet memory constraints and improve performance. Both execution–time profiling tools and tools for assessing the impact of configuration decisions should be included.

Other aspects of the CPE we emphasize include a *librarian* tool to make the use of large libraries of generic components more convenient through a combination of traditional data base, hypertext, and graphical display capabilities; an *interpreter* and/or *incremental compiler* that supports interoperability between Æ and Ada code; a *persistent object base* that provides the core facilities for sharing information among tools and supports strict configuration control as required in many defense related applications; and *inference tools* supporting consistency checking and direct execution

of meaning declarations.[3]

Research team This proposal brings together a research team with many years of experience in programming language and programming environment design; advanced programming methodology in many languages including Ada, C, C++, CLU, Lisp, Scheme; and formal methods and tools for reasoning about the consistency, completeness, and correctness of software requirements, designs and implementations. Particularly relevant to this project is our experience in developing generic libraries of software components in Ada, C++, and Scheme [22,23,24,18], from which we gained many of the key insights behind our approach to the design of the structural prototyping constructs of Æ. Our extensive experience with both theory and practical applications of formal methods [16,17,4,3], including development of transformation techniques for synthesizing parallel distributed Ada programs from a limited class of conditional equations that define recurrence relations [31,30,19], will also enable us to develop the necessary support for an advanced, but tractable, approach to behavioral prototyping.

The rationale for and some of the technical issues related to the language extensions are discussed in Section 2. We believe that when completed the design of Æ will not only be a good pragmatic approach for the four year time frame envisioned by DARPA, but will also provide a solid foundation for many years to come for advanced formal methods of prototype (and program) analysis and transformational implementation.

2 Technical Discussion

2.1 Basic semantic concepts and mechanisms of Æ

We propose to design the Æ language to support structural prototyping in a manner that is competitive with or exceeds languages like Common Lisp and Smalltalk in the ease and speed with which prototypes can be built, even while allowing more static checking and analysis and more support for generic library components and higher order programming methods than Ada itself provides.

Furthermore, Æ will strongly support behavioral prototyping. Each major abstraction mechanism in Æ will have an associated *meaning* construct. Although we leave the use of meaning declarations optional, we are attempting to make their use as simple and productive as possible

[3] Except for the nature of the inference tools, the CPE is not further discussed in this proposal, but environment requirements and issues will be detailed in a chapter of Initial Language Design document to be delivered.

during the behavioral prototyping stage that should precede structural prototyping.

Perhaps most importantly, Æ will permit smooth transitioning of structural prototypes into deliverable systems in Ada, primarily because it supports construction and use of generic libraries at both the prototyping and production levels. Behavioral prototypes expressed as meanings can also be carried through to the production system, serving as formal documentation of behavior and a contract for the implementors and maintainers of the production system.

We propose additional language mechanisms in five areas:

- Generic library organization and semantics
- Higher order procedures and functions
- Dispatch types and run–time dispatching
- Inheritance
- Parallel and distributed computation

For each of the major abstraction mechanisms introduced, as well as for those of Ada itself, there will be a meaning construct for associating behavioral semantics with the mechanism. We now sketch the way we expect Æ to provide these mechanisms.

2.1.1 Generic library organization and semantics

The primary constructs we introduce in this area are *form*, *form meaning*, and *package meaning*. A form is syntactically the same as a package specification, except that the keyword `form` replaces `package`. A form denotes the family of all package specifications that *match* the form, in the sense that they contain basic declarative items identical to those in the form or which can be *fitted* to those in the form (a matching package specification can also contain other basic declarative items). Fitting is a kind of name correspondence and can be defined formally as in [4,25,26], for example.

The main use of a form is as a (new kind of) parameter to a generic package. Such a parameter can be instantiated with any package that matches the parameter. For example,

```
form Preorder is
      type Element is private;
      function "<="(E1, E2: Element) return Boolean;
end Preorder;
generic
      with package P: Preorder;
```

```
package List_Sorter is
    this specification and/or the package body (not shown) would
    contain references to basic declarative items appearing in P
end List_Sorter;
package Alpha is
      type Alphabetic is private;
      function Lex_Order(A, B: Alphabetic) return Boolean;
       . . . other functions, etc
end Alpha;
package Alpha_List_Sorter is new
  List_Sorter(P => Alpha as Preorder(Element => Alphabetic, "<=" => Lex_Order));
```

In this case the fitting is expressed explicitly, but in many cases all or part of the fitting could be omitted because of default rules. (Such rules have been defined in [4].)

In Ada, this example could also be done by making "<=" itself be a parameter to List_Sorter. In other cases, such as when a form contains generic units, there is no counterpart in Ada since parameters to a generic package may not be generic units.

A meaning can optionally be associated with a package, providing *properties* for the procedures, functions and constants of the package. The properties that may be specified include descriptors for operations such as commutative, associative, reflexive; and *axioms*, which are conditional equations expressing relations between functions or procedures of the package. Similarly a meaning can be associated with a form; this places a stronger requirement on matching: the meaning of a package that matches the form must imply the meaning given to the form. E.g., with

```
form Preorder meaning is
          "<=" is reflexive, transitive;
end Preorder;
package Alpha meaning is
          Lex_Order is reflexive, transitive, total;
end Alpha;
```

the Preorder form is still matched by the Alpha package.

Use of forms as parameters to generic packages allows a system designer to build up package requirements incrementally (using the extended private inheritance mechanism described in Section 2.1.4) and organize them in semantically meaningful ways. For example,

```
form Total_Order meaning is
```

```
    change Preorder meaning add
          "<=" is total;
end Total_Order;
```

Then `Lex_Order` matches `Total_Order` as well as `Pre_Order`. Alternatively, one could write

```
form Total_Order meaning is
    change Preorder meaning add
          forall E1, E2: Element
               axiom: (E2 <= E1) if not (E1 <= E2);
end Total_Order;
```

Then `Lex_Order` would still match `Total_Order` because internally the matcher would interpret the occurrence of `total` in the meaning of `Lex_Order` as equivalent to the axiom shown (one might also have given the axiom, or a different axiom that implies it, in `Lex_Order`). Thus, although some special properties are singled out for convenience as descriptors that can be matched syntactically, in general the matching performed on meanings is a semantic process supported by formal reasoning.

The combination of form and form meaning constructs is similar to mechanisms used to specify parameter requirements in several specification languages—e.g., OBJ *theories* [8,9], Axis *PROPS* [4], some forms of Larch *traits* [11]; it is also similar to *categories* in the Scratchpad II programming language [12]. In Scratchpad II, categories have been used to organize a complex hierarchy of algebras, such as rings, integral domains, fields; these categories serve as requirements on parameters to domain constructors such as `Polynomial(R: Ring)`. In working out the details of the design of forms and generic packages with form parameters, we expect to draw heavily on this previous language experience as well as our own experience with constructing generic libraries in Ada [23] and using formal methods of semantic matching [17,16,4].

2.1.2 Higher order procedures and functions

Based on our experience with developing generic libraries of software components in Ada, C++, and Lisp, we strongly advocate the use of higher order procedures; i.e., procedures that take other procedures as arguments or return procedures as their results. While generic software libraries can be built without using higher order techniques, we have found they are a key means of achieving extensive code reuse, even within the libraries themselves—as illustrated for example in [18] and [23].

Some forms of higher order procedures are already supported in Ada via generics, and, in comparison with Lisp, can result in highly efficient code, since no run–time type checking is required

and much of the layering of procedure calls can be removed by inlining. We propose to provide language constructs in Æ to make these simple cases more convenient and to provide for cases that cannot be done directly with generics, such as procedures that take second order procedures as parameters. In fact, the form construct described on page 5 is a solution to the latter problem, since in Æ packages and subprograms may have form parameters and forms may contain generic units. We also intend to allow anonymous instantiations. These are relatively minor language extensions, but experience has shown that this kind of limited support for higher–order techniques is sufficient in practice [10,24].

2.1.3 Dispatch types and method dispatching

The dispatch type facility imports into Æ the key technology from object oriented programming languages. The most essential difference between an Æ structural prototype and an Ada production program derived from it will be the extensive use of dispatch types and run–time method dispatching in the Æ prototype to achieve flexibility, localization of code, and reuse of previous designs; "freezing" a dispatch type at some point allows automatic transformation into standard Ada code using variant records and case statements, with substantial optimizations being enabled in many cases.

We propose to extend Ada with run–time method dispatching based on single type dispatch. This facility allows heterogeneous data structures and allows code to be localized near the corresponding types with the proper code being selected at run–time. It also allows multiple views of the same data.

The main idea to achieve method dispatch is to add explicit "dispatch types" to Ada and to bundle arbitrary values together with a dispatch type tag in a "view type" definition. A dispatch type tag is like a value, but it cannot be stored in a variable. A pair (value, dispatch type tag) is called an object, and access to its value component is done only by operations accessible via the dispatch type tag.

A dispatch type is a named set of procedure and function headers. The first argument of each header must be a parameter of that dispatch type, where the mode is arbitrary. For example,

```
dispatch type SHAPE is
        procedure MOVE(OBJ: in out SHAPE; X, Y: REAL);
        function  AREA(OBJ: in SHAPE) return REAL;
        procedure DISPLAY(OBJ: in out SHAPE);
end SHAPE;
```

Semantically, the dispatch type `SHAPE` denotes a parameterized record structure containing

a data slot and operation slots where (a) occurrences of SHAPE can be actualized by an actual type name, and (b) the operation headings can be actualized to actual operations of compatible signature. A value of a dispatch type can be viewed as a virtual function table a la C++: for each operation of SHAPE there is a corresponding field containing a pointer to a function or procedure.

Run–time dispatch A variable of dispatch type allows run–time dispatch. For example, if used as formal parameter of some operation,

```
procedure DIAGONAL_MOVE(S: in out SHAPE; X: REAL)
begin
        MOVE(S, X, X);
end DIAGONAL_MOVE;
```

the code created for the call `MOVE(S, X, X)` reflects the fact that the first parameter is of dispatch type: the value passed for `S` contains a `MOVE` field, and its dereferenced pointer is applied to `(S.data, X, X)`.

Note that in places other than formal parameters of procedures or functions, a variable of dispatch type SHAPE has to be initialized, according to Ada conventions.

View types A view type is obtained from a dispatch type by supplying concrete definitions for the operations and by binding the data slot to a particular type. Assume:

```
type      SQUARE is record L_Corner: POINT; Length: REAL end;
procedure DISPLAY_SQUARE (X: in out SQUARE) is begin ... end;
function  AREA_SQUARE (S: in SQUARE) is begin ... end;
procedure MOVE_SQUARE (S: in out SQUARE; X, Y: REAL) is begin ... end;
```

A view type based on SQUARE and the listed operations is achieved by

```
type OBJECT_SQUARE is view SQUARE as SHAPE
        with DISPLAY_SQUARE, AREA_SQUARE, MOVE_SQUARE;
```

The operator `view_as_with` constructs a record–like structure from a syntactic correspondence

```
(SHAPE,DISPLAY,AREA,MOVE) to (SQUARE,DISPLAY_SQUARE,
                              AREA_SQUARE, MOVE_SQUARE).
```

SQUARE is called the base type of OBJECT_SQUARE, and SHAPE is the dispatch type of OBJECT_SQUARE. If meanings are supplied, then a semantic check can also be performed.

A variable of a view type is created by supplying a value for the base slot which is copied into the resulting record value.

```
Square_1: SQUARE := Some_Square;
Object_Square_1: OBJECT_SQUARE := OBJECT_SQUARE(Square_1);
```

The initialization for `Object_Square_1` uses the conversion function associated with `OBJECT_SQUARE`. Conversions are automatically generated. Values of view types are objects.

There are four things one can do with view types: (1) convert base type values to view type values (attaching dispatch information); (2) convert in the opposite direction (stripping dispatch information); (3) pass view type values as dispatch type arguments and (4) assign them to dispatch type variables.

Freezing a dispatch type There is also a `freeze` pragma that can be applied to an dispatch type, the effect of which is to disallow creation of new view types related to that dispatch type and thus to permit procedures and functions expressed in terms of the dispatch type to be transformed into inline efficient code using a case statement (which then can be optimized). E.g., the body of the `MOVE` procedure discussed on page 9 would be replaced by a case statement, with the code for `MOVE_SQUARE` becoming one arm, and the code for corresponding `MOVE` procedures in other views of `SHAPE` becoming other arms. Note that "freezing" is in effect a mechanism used in the transition from prototype to a production version, not just a notion on the conceptual level (like frozen design).

Orthogonality Dispatch types and view types are like standard Ada types in that they can be used in any context in which Ada types can be used. In particular, there are generic dispatch types, possibly with formal dispatch type parameters, and there are access types of view types.

The decision to create view types first, and then use them in object declarations, follows the Ada philosophy of disallowing implicit types.

2.1.4 Inheritance

Inheritance of dispatch type operations One kind of inheritance provided is a consequence of the dispatch type mechanism: operations such as `DIAGONAL_MOVE` on page 9 are inherited by any view type of the dispatch type. Thus dispatch types and view types may be used to construct a kind of class hierarchy. Since multiple views of the same data are allowed, we have a kind of multiple class inheritance.[4]

[4]Note that the procedures and functions of the dispatch type itself are *not* inherited, since they are only specified in the dispatch type and implementations must be supplied in the view type.

A limited form of (single) class inheritance can be expressed in Ada itself, using generic packages. E.g., a SHAPE generic package could be defined with MOVE, AREA, and DISPLAY as generic parameters and containing procedures such as MOVE_DIAGONAL. An instantiation of SHAPE with SQUARE operations then makes MOVE_DIAGONAL available on SQUAREs. Note however that this approach does not allow run–time selection of operations, as dispatch types do.

These kinds of inheritance do not allow redefinition of inherited operations, but the following mechanism does.

Extended private inheritance We intend to provide a general purpose reuse mechanism which is an extended private inheritance. It is extended, with respect to object oriented languages, because it is applicable not only to types but to other language constructs such as packages or procedures. It is private because the fact that the new type (or package, procedure, etc.) is inherited from an old type (and not produced manually) is not visible; keeping inheritance private is critical to permitting its use in constructing reusable software components [28,29]. It is a generalization of the Ada derived type mechanism. For example, if a `Linked_List` package is defined with nodes containing a single link field, a `Doubly_Linked_List` package can be obtained by reusing the `Linked_List` package specification without modification, via a `copy` construct, and by reusing the package body with minor additions and changes expressed with `add` and `change` constructs.

The `copy` construct copies a named Ada structure, while `change` copies and modifies, for example by `adding`, `deleteing`, or `replaceing` syntactically appropriate components. These mechanisms will also be provided for other Ada constructs (e.g., procedures, blocks, statements).

By themselves, the mechanisms just described do not provide enough control over semantics; for example, it is possible to `change` a structure in nonsensical ways. By associating meaning declarations with the constructs, however, we can support the logical notion of conservative extension (cf. [13,8,4]).

2.1.5 Parallel and distributed computation

To enable prototyping of parallel and distributed systems, the following mechanisms will be defined in Æ: automatic definition of tasks in forms and packages; alternative interprocess interaction primitives; extended semantics of shared variables; and incremental prototyping by environment simulation. A set of conditional equations defining the meaning of a form or package will also be used to determine data dependencies between elements of this form (package). If a parallel or distributed implementation is desired, the data dependencies will be used to define the parallel components through grouping form (package) elements into tasks in a manner similar to that in [31].

Whenever appropriate, the procedure or function call will be transformed into a pair of interprocess interactions: the first to request the execution of the called procedure (function) and to provide input parameters, and the second one to receive the results (output parameters).

In addition to Ada rendezvous, Æ will support other interprocess interaction primitives through the generic libraries. In particular, we are interested in the message–passing primitives *non–blocking send* and *blocking receive*. As discussed in [30] there are simple transformations which substitute non–blocking send and blocking receive for rendezvous, and vice versa, with varying degrees of efficiency. The customer/server relationship can be often described more naturally using non–blocking send and blocking receive than rendezvous. For example, if a procedure `MOVE(OBJ: in out SHAPE; X,Y: REAL)` can be run in parallel with the calling procedure, then the call:

```
MOVE(CURRENT_OBJECT, POSX, POSY);
```

can be transformed into:

```
SEND(MOVE, CURRENT_OBJECT, POSX, POSY);
   . . . computations not using CURRENT_OBJECT
RECEIVE(MOVE, CURRENT_OBJECT);
   . . . computations using CURRENT_OBJECT
```

There is no need to delay the calling procedure at the point of `SEND`, even if `MOVE` is not ready to accept the message (assuming some queueing capabilities are provided by the implementation). The calling procedure will be executing statements that do not involve the result of the call. The opposite is also true; i.e., there is no need for the `MOVE` to wait until the caller is ready to accept the result: `CURRENT_OBJECT` can simply be queued at the calling procedure port. However, the computation cannot continue beyond the `RECEIVE` without getting the new value of `CURRENT_OBJ`. Thus, non-blocking `SEND` and blocking `RECEIVE` are a more appropriate implementation here than a pair of rendezvous.

Data dependencies are crucial in determining which components can run in parallel. However in some situations designers are interested in using data simultaneously in several parallel components. To disregard data dependency between functions and procedures, the corresponding data structure will have to be denoted as shared. For example, calls to procedures:

```
MOVE(CURRENT_OBJ, POSX, POSY); DISPLAY(CURRENT_OBJ);
```

are dependent through the data structure `CURRENT_OBJ` and therefore cannot be executed in parallel. However, if the `CURRENT_OBJ` is shared then both can run in parallel with the only danger being of displaying an "old" position of the object.

Æ will provide the tools for simulating the unimplemented parts of the parallel or distributed environment through parallel component stubs. The stub will consist of the delay function which would simulate execution time and the sequence of the returned values which will be produced in response to the calls. Implemented generic library components, executable conditional equations describing meanings, and stubs simulating unimplemented packages can be mixed freely to enable designers incremental development of the designed systems.

2.2 Formal methods for prototype analysis and transformation

The *meaning* constructs in Æ can be used in several important ways for behavioral prototyping:

Executability of behavioral prototypes Meanings can be used not only to express intended behavior of operations for purposes of documentation, but also in many cases they are *executable*, through treatment of conditional equations as *rewrite rules*. Although not efficient enough for production use, such rewrite rule implementations can provide immediate and useful feedback about functional behavior. When behavior is observed to be different from what was intended, one can adjust the meaning descriptors and equations, before investing the greater effort required to develop a structural prototype.

One can continue to use the equational implementation for some operations while developing meanings for other operations, or even while developing structural prototypes or production implementations of other operations—we will make the rewrite rule interpreter interoperable with the normal interpreter and with compiled code.

We will draw heavily from our experience in building the Affirm verification system as well as from our recent work on RRL (Rewrite Rule Laboratory), a theorem prover based on rewriting and completion. We have successfully used term rewriting methods and completion procedures for analysis of specifications, checking for consistency and sufficient completeness of equational axiomatizations, inductive and deductive reasoning [21,15].

User definable optimizations Meanings can also be used to allow user definable optimizations, where certain rewrite rules are viewed as source level optimizers for the library packages. This will allow the definition of many standard optimizations on user defined types and operations.

Libraries of reusable theories Experience with the Ada Generic Libraries [23] suggests considerable advantage will be gained by developing libraries of *reusable theories* (generic meanings) to accompany generic software components. These theories can be used to verify properties of generic components and support a methodology for reliable program construction based on genericity, as

discussed in [24]. Once proven correct, these components can be instantiated and repetitively used for many diverse applications, the only additional proof required being that the actual parameters satisfy axioms assumed by the proof at the generic level. We have also been investigating the use of theory schemes to structure proofs, which will aid in reasoning about higher order procedures. We will continue these investigations by developing a set of reusable theories for the kernel families described in Section 2.3, using the Æ constructs for organizing meanings.

Working in RRL, Shang [27] implemented meta–operators on concepts and theories (as discussed in [13]; see also [8]) and experimented with libraries of generic modules. This experience will be helpful in implementing reuse mechanisms such as **add** and **change** for supporting inheritance on meanings.

Automated consistency and reasonableness checking Meanings permit semantic consistency and reasonableness checks while instantiating generic packages, defining view types from dispatch types, and operating on meanings using **add** and other inheritance mechanisms for reuse. Methods discussed in [21,15] can be used for checking consistency and definitional completeness of equational axiomatizations. Whenever a correspondence is made between linguistic constructs, as in definition of a view type or instantiation of parameters to generics, term rewriting based inference methods can be used to perform limited semantic checking. For instance, when package `Alpha_List_Sorter` is defined as an instance of a generic `List_Sorter`, the requirement that `Lex_order` be reflexive and transitive can be derived from its meaning. The check for this particular example is straightforward, but in other cases, such a semantic check can require nontrivial deductive and inductive reasoning. Our implementation of the completion procedures and the cover set method for induction in RRL is capable of semi–automatically performing such nontrivial semantic checks.

Support for a transformational approach A longer term potential use of equational implementations is in support of program transformations and semi–automated derivation of production code. Since behavioral equations capture the semantics of components more completely than syntactic and type information and in a more tractable form than code bodies, they can support more extensive use of semantic information in choosing which transformations to apply at a given stage of automated derivation.

Term rewriting techniques have been found useful in program transformation, especially in generating equational implementations from equational specifications [14,5]. We will also explore the use of term rewriting techniques, in particular the Knuth–Bendix completion method and inductive reasoning, for generating code using generic library components and higher–order functions from

meanings.

An important area in which we have experience with transforming equational implementations into efficient production code is the subclass of algebraic equations that define recurrence relations [31]. We will explore progressive refinement of equational description of meanings into this restrictive class of recurrence equations.

2.3 Generic library development

One of the main reasons that Lisp became a success as a prototyping language is that it has a large built–in set of data manipulation facilities. We regard it as fundamentally important to provide similarly comprehensive libraries of data structures and operations on them for use during prototyping in Æ; many of these capabilities should also be provided in Ada libraries for production use.

The selection and structure of the facilities in Common Lisp evolved over time in an ad hoc manner; we can now see how to structure libraries with greater modularity and potential for reuse, taking advantage of the generic units feature of Ada and its extensions in Æ.

Families Our main idea for the organization of the libraries is the notion of *family*. A family is a set of related data structures that can be converted into each other and that share a common meaning. Every family has three component parts: a common set of operations, a set of data structures, and a set of generic algorithms.

Of course, not every operation on the family is directly implementable on a particular data structure in the family. Also, operations can be classified according to some common invariants.

Sequences The most useful family to have is that of sequences. It has a very rich set of operations such as: inserters (Insert, Append, Drop, etc.); permuters (Reverse, Random_Shuffle, etc.); partitions (Delete, Partition, Stable_Partition, etc.); searches (Search, Mismatch, Find, etc.). It has several data structures: singly–linked lists; doubly–linked lists; vectors; extensible vectors. Dozens of useful sequence algorithms are known, many of which can be represented in a generic form, that will implement different operations on different data structures in the family.

Other families We intend to provide several other commonly used families, such as stacks, queues, priority queues, trees, dags, graphs. One of the most challenging tasks would be to develop a generic high level I/O library (generic windows).

References

[1] Baker, T., "Ada Runtime Support Environments to Better Support Real–Time Systems," in [2].

[2] Barnes, et.al., *Proc. of the International Workshop on Real–Time Ada Issues*, Moretonhampstead, Devon, UK, 13-15 May 1987.

[3] Bruno, J., Szymanski, B. (1988), Conditional Data Dependence Analysis in an Equational Language Compiler, Proc. of *Third International Conference on Supercomputing*, Boston, MA, May, 1988.

[4] Coleman, D., et.al. (1988). *The Axis Specification Language.* Technical Report HPL-BRC-TM-88-020, Software Engineering Dept., Hewlett–Packard Laboratories, Bristol, U.K., 1988.

[5] Dershowitz, N. (1985). Computing with rewrite systems. *Information and Control* 64, 122-157.

[6] *Draft report on requirements for a common prototyping environment.* Ed. Gabriel, R.P., Nov. 11, 1988.

[7] Firth, R. "A Pragmatic Approach to Ada Insertion," in [2].

[8] Goguen, J. (1984). Parameterized programming. *Transactions on Software Engineering*, SE-10(5), 528-543, September 1984.

[9] Goguen, J. (1986). Reusing and interconnecting software components. *IEEE Computer*, Feb. 1986.

[10] Goguen, J. (1988). *Higher–Order Functions Considered Unnecessary for Higher–Order Programming*, SRI Technical Report SRI–CSL–88–1R, April 1988.

[11] Guttag, J.V., Horning, J.J., and Wing, J.M. (1985). *Larch in Five Easy Pieces*, Digital Systems Research Center, Technical Report 5, July 1985.

[12] Jenks, R.D., Sutor, R.S., and Watt, S.M. (1986). Scratchpad II: An abstract datatype system for mathematical computation. IBM Research Center Report RC12327, Yorktown Heights, NY, Nov. 1986.

[13] Kapur, D., Musser, D.R., and Stepanov, A.A. (1982). Operators and Algebraic Structures. Proc. of *1981 Conference on Functional Programming Languages and Computer Architecture*, Oct. 1981, New Hampshire, 59-64.

[14] Kapur, D., and Srivas, M. (1985). Rewrite rule based approach for synthesizing abstract data types. Proc. of *Colloquium on Trees in Algebra and Programming (CAAP)*, Berlin, March 1985, LNCS 185, Springer Verlag, 188 - 207.

[15] Kapur, D., Narendran, P., and Zhang, H. (1986). Proof by induction using test sets. *Eighth International Conference on Automated Deduction (CADE-8)*, Oxford, England, July 1986, Lecture Notes in Computer Science, 230, Springer Verlag, New York, 99-117.

[16] Kapur, D., and Zhang, H. (1987). *RRL: A Rewrite Rule Laboratory - User's Manual.* GE Corporate Research and Development Report, Schenectady, NY, April 1987.

[17] Kemmerer, R., ed. (1986). *Verification assessment study final report, Volume III, The Affirm system.* National Computer Security Center, Fort George G. Meade, Maryland.

[18] Kershenbaum, A., Musser, D.R., and Stepanov, A.A. (1988). *Higher order imperative programming.* Computer Science Dept. Rep. No. 88-10, Rensselaer Polytechnic Institute, Troy, New York, April 1988.

[19] I. Lee, N. Prywes, B. Szymanski (1986), Partitioning of Massive/Real-Time Programs for Parallel Processing, chapter in: *Advances in Computers*, vol. 25, 1986, pp. 215-275.

[20] Musser, D.R. (1980). On proving inductive properties of abstract data types. Proc. *7th Principles of Programming Languages (POPL).*

[21] Musser, D.R., and Kapur, D. (1982). Rewrite rule theory and abstract data type analysis. *Computer Algebra, EUROCAM, 1982* (ed. Calmet), Lecture Notes in Computer Science 144, Springer Verlag, 77-90.

[22] Musser, D.R., and Stepanov, A.A. (1987). A library of generic algorithms in Ada. *Proc. of 1987 ACM SIGAda International Conference*, Boston, December, 1987.

[23] Musser, D.R., and Stepanov, A.A. (1988). *Ada Generic Library Linear Data Structure Packages*, Volumes 1 and 2, General Electric Corporate Research and Development Reports 88CRD112 and 88CRD113, April 1988. To be published in book form by Springer-Verlag, 1989.

[24] Musser, D.R., and Stepanov, A.A. (1988). Generic programming. Invited paper, to appear in *Proc. 1988 International Symposium on Symbolic and Algebraic Computation, Lecture Notes in Computer Science*, Springer-Verlag, July, 1988.

[25] Olthoff, W., "The Module Concept of ModPascal: Integration of Abstract Data Types in an Imperative Programming Language," *Proc. Workshop on Software Architecture and Modular Programming*, Teubner Verlag, 1986, pp. 123-137.

[26] Olthoff, W. (1986), "Augmentation of Object-Oriented Programming by Concepts of Abstract Data Type Theory: The ModPascal Experience," *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, sl SIGPLAN Notices, Vol. 21 (11), 1986, pp. 429-443.

[27] Shang, W. (1989). A hierarchical language with semantic checking. Forthcoming M.S. Thesis, Lab. for Computer Science, MIT, 1989.

[28] Snyder, A. (1986) "Encapsulation and Inheritance in Object-Oriented Programming Languages," OOPSLA-1986.

[29] Snyder, A. (1987) "Inheritance and the Development of Reusable Software Components." In *Research Directions in Object-Oriented Programming*, edited by Bruce Shriver and Peter Wegner, MIT Press, 1987.

[30] Szymanski, B. (1988). "Beyond ADA - Generating Ada Code from Equational Specifications," Proc. of *6th Annual National Conference on ADA Technology*, Washington D.C., March, 1988.

[31] Szymanski, B. (1987). "Parallel Programming with Recurrent Equations," *International Journal on Supercomputer Applications*, Vol. 1, No. 2, pp. 44-74, 1987.