

COMPILING KNOWLEDGE-BASED SYSTEMS TO ADA: THE PRKADA CORE

Robert E. Filman^{*} and Paul H. Morris[†]
IntelliCorp, Inc.
1975 El Camino Real
Mountain View, California 94040
USA

Received 17 May 1996
Revised 9 October 1997

This paper describes the implementation of PrkAda, a system for delivering, in Ada, Artificial Intelligence and object-oriented applications developed using the ProKappa system. (ProKappa is a modern, multi-paradigm knowledge-based-system development tool. It includes facilities for dynamic object management, rule-based processing, daemons, and graphical developer and end-user interfaces. ProKappa is a successor system to KEE.) Creating PrkAda required creating a run-time, Ada-language, object-system “core,” and developing a compiler to Ada from ProTalk (ProKappa's high-level, backtracking-based language). We describe PrkAda ProTalk compiler in a companion paper [5]. This paper concentrates on the issues involved in implementing an AI application delivery core, particularly with respect to Ada, including

- Automatic storage management (garbage collection) without either the cooperation of the compiler or access to the run-time stack,
- Dynamic (weak) typing in a strongly-typed language,
- Dynamic objects (objects that can change their slots and parentage as the program is executing)
- Dynamic function binding in a language designed to preclude “self-modifying programs,” and
- Implementation trade-offs in object-oriented knowledge-based systems development environments

Keywords: Knowledge-based systems compilation, knowledge-based systems development tools, Ada, ProKappa, object-oriented systems, application delivery environments

^{*} Current address: Advanced Technology Center, Lockheed Martin, 3251 Hanover Street O/H1-41 B/255, Palo Alto, California 94304, filman@ict.atc.lmco.com

[†] Current address: NASA AMES Research Center, M/S 269-1, Moffitt Field CA 94035, pmorris@ptolemy.arc.nasa.gov

1. Introduction

Artificial Intelligence as a technology has arisen because it has proven difficult to construct, using conventional software technologies, programs that solve certain classes of problems. These problems are characterized by

- An irregular structure,
- The necessity of creating complex data structures and applying semantically rich interpretations to these structures, and
- The usefulness of a large library of structure manipulation routines together with an environment that can manipulate, coherently present, and easily navigate such structures.

In AI, often the problem and its possible solutions are not well understood at the beginning of the development process. Such problems take exploratory programming to reveal the representational and heuristic possibilities and problems.

Ada¹ is an interesting target for AI system development. Ada had a principled design centered on the software development and maintenance process (at least as it was understood in the late 1970s). Many features of Ada illustrate the influence of this methodology, including packages (Ada's abstract datatype mechanism), overloading (a structured, compile-time form of polymorphism), generics (a structured form of function and type substitution), language restrictions to prevent certain classes of errors (e.g., the inability to get pointers to object on the stack; limited private types), and a rich but restricted language of types (one can create derived types that inherit functionality, but cannot create function-valued functions). In implementation terms, an Ada compiler takes pains to eliminate dynamic application decisions and certain potential inconsistencies. Issues that cannot be decided at compile-time (e.g., the type of an entity) result in compilation errors.

Classical software engineering principles imply that the requirements, specification and design of a system be essentially complete before beginning its implementation (e.g., waterfall models). Ada is a language meant to embody and enforce those principles. Ada works best when used with a methodology where the programmer defines the modules, behaviors and “kinds of things to be processed” early in the application development process, and progresses by completing these abstractions.

In theory, Ada provides mechanisms (such as packages) to shield the later parts of the development from changes in the implementation of the foundation. However, our experi-

¹ Ada, as used in this paper refers to Ada83. The issues discussed (e.g., storage management of dynamic objects, alternative data structures for AI object systems) are especially pertinent to languages of its generation (e.g., C, Pascal). Some of these issues (e.g., storage management) are more easily treated in languages like C++, Java and Ada95 that implement allocators and deallocators. We discuss the differences below.

ence has been that the abstraction barrier in Ada is too permeable.² Lisp, with its weak-typing, consistent syntax, and function-valued objects avoids these pitfalls.

Ada's formal structure makes AI-like experimentation (or systems developed using AI flexibility) difficult. Similarly, there is a wealth of programming environment support (such as symbolic debuggers, read-eval-print loops, and generalized pretty-printing functions) that AI environments have and that Ada environments lack.

1.1. AI development environments

What makes a good AI development environment? The best environments for AI-like exploratory programming are knowledge-based systems (KBS) development tools. These tools provide:

- 1. Objects.** Objects represent the elements of the domain. Objects have slots that describe their properties, form class/instance hierarchies, inherit values along these hierarchies, have daemons that invoke behavior on slot access and modification, and compute through messages (i.e., object-oriented programming). Languages such as C++ provide objects as a mechanism for programmer control over the layout and organization of data structures, including structures that include pointers to functions. In KBS tools, objects are an ontological commitment about the way a domain ought to be modeled, and incorporate a system-defined implementation and associated set of utilities.
- 2. Inference engines.** Inference engines provide a mechanism for expressing the conclusions to be drawn and the actions to be taken during system execution. Inference engines are often rule systems.
- 3. Graphical development environments.** A graphical development environment provides the KBS developer with tools for understanding and modifying knowledge base structures and program behavior.
- 4. Application graphics tool kits.** Application graphic tool kits facilitate creating the end-user application graphics. Application graphics tool kits typically provide a collection of "widgets" that range from simple value displayers and menus to tables and node/arc-graphs.

² Examples of this permeability include (1) a mapping implemented as an array can be a parameter to a subprogram. A mapping implemented as a function cannot. (2) a generic package used with a private type cannot be used for a limited private type, even though the generic body may not do assignment (or otherwise violate the limitations of the limited type). (3) A subprogram that matches a generic subprogram parameter and then changes to acquire an additional parameter can no longer be used to instantiate that generic, even if the additional parameter has a default value. However, the altered subprogram is usable in every other context (barring overloading resolution ambiguities). (4) Changing the specification of basic types can require overall system recompilation.

1.2. System goals

The earliest, most prominent, and most powerful commercial KBS development tools were written in Lisp. Examples of such systems include KEE [1], ART [2], and KnowledgeCraft [3]. These systems enhance the native Lisp environment, yielding a synergistic synthesis of symbolic representation and programming, automatic storage management (garbage collection) and an extensive set of native graphic and symbolic debugging tools. Unfortunately, such Lisp environments have been less than complete commercial successes. Problems faced by these products include dependence on an unpopular programming language, a lack of connectivity to and embeddability within conventional systems, the requirement of a large run-time environment, and the difficulty of doing real-time programming with most implementations of garbage collection. These limitations restricted most KBS implementations to be either laboratory prototypes or (relatively) stand-alone applications.

Currently, there is a trend towards the development of KBS tools in “more conventional” languages. Most such efforts are C-based. C has several advantages. It is syntactically simple (and gives the illusion of semantic simplicity). C is easy to implement and thus runs on almost all platforms. Programmers who know C are easy to find, C having emerged as something of a lingua franca of programming. Similarly, C is easily learned (though not necessarily easily mastered). Finally, C appeals to the subconscious assembly language programmer, providing a straightforward mapping between high-level constructs and their low-level implementation and allowing access to primitive machine operations such as the addresses of functions and pointers to the run-time stack. On the other hand, C has the disadvantages of being relatively unstructured, having few built-in language features to deal with the problems of building large systems, of having no native debugging environment, of being relatively non-standardized (and thus nontrivial to port), and of being difficult to understand and maintain.

In response to the ebb in demand for Lisp-based tools and the corresponding swell of C, commercial, C-based KBS tools are emerging. This work was based on one such tool, ProKappa [4]. ProKappa can be viewed as an attempt to implement the features of Lisp-based tools like KEE in C. ProKappa includes dynamic objects, garbage collection, and appropriate graphics. While ProKappa alleviates many of the problems of C-language development, it nevertheless runs on only a limited set of platforms and fails to satisfy the requirements of certain clients for Ada-based systems

The goal of this effort was to create a delivery environment for ProKappa applications in Ada. That is, we wanted a direct path for taking applications created in the ProKappa development environment and delivering them in Ada. Development would include creating a knowledge base of objects, a set of C and Ada-language methods, and a collection of rules and code in the ProKappa language ProTalk. The ProTalk compiler [5] would then translate the user's ProTalk to Ada. The user's C-language message handlers

could be manually translated into the equivalent Ada calls (a straightforward task, there being a one-to-one correspondence of almost all datatypes and functions between the two systems). The complete Ada system would be composed of our Ada-language core object library, the compiled ProTalk, the user's message code and the embedding system. The core object library can, at run-time, read and create ProKappa knowledge bases. The net result is an AI application delivered in a pure-Ada environment. We illustrate this process in Figure 1.

1.3. System overview

Given the resources available for this effort, we did not attempt to duplicate the entire ProKappa environment in Ada. Rather, we developed enough of the system to enable embedded applications: an Ada-language *core*—the primitive datatypes and object manager and a compiler from ProTalk functions to Ada. We have also demonstrated the ability to write simple methods in Ada and use them in the ProKappa C environment [6]. This work includes no graphics, either developer or end-user, and only the most minimal developer environment—the premise being that an application would be developed in conventional ProKappa C and then compiled/ported to the Ada environment.

The Ada-language core is a collection of Ada packages, generics, and subprograms that implements the delivery functionality of the ProKappa substrate and object manager. (A more detailed description of the substrate and object manager can be found in [7].) Important components of this core include:

- 1.** The basic datatypes definitions for ProKappa types (PrkTypes), including structures and routines for objects, symbols, lists, and arrays. ProKappa, like Lisp, is weakly typed. Thus, to Ada the ProKappa datatypes must appear simultaneously to be of the same type but nevertheless differentiable.
- 2.** A mechanism that, when used with the appropriate hygiene, provides automatic storage management (garbage collection) of (to-be-collected) PrkTypes.
- 3.** A set of functions that implement the semantics of ProKappa objects, including the ability to dynamically create objects with multiple parents, find objects by name, modify and retrieve slot and facet values, and dynamically inherit slots, slot values, facets, and facet values.
- 4.** Monitors (daemons) that can be invoked on slot access or modification.
- 5.** The ProKappa style of object-oriented programming with dynamically varying instance methods. These allow (a representation of) a function to be a slot value. Messages sent to objects about that slot invoke that function.

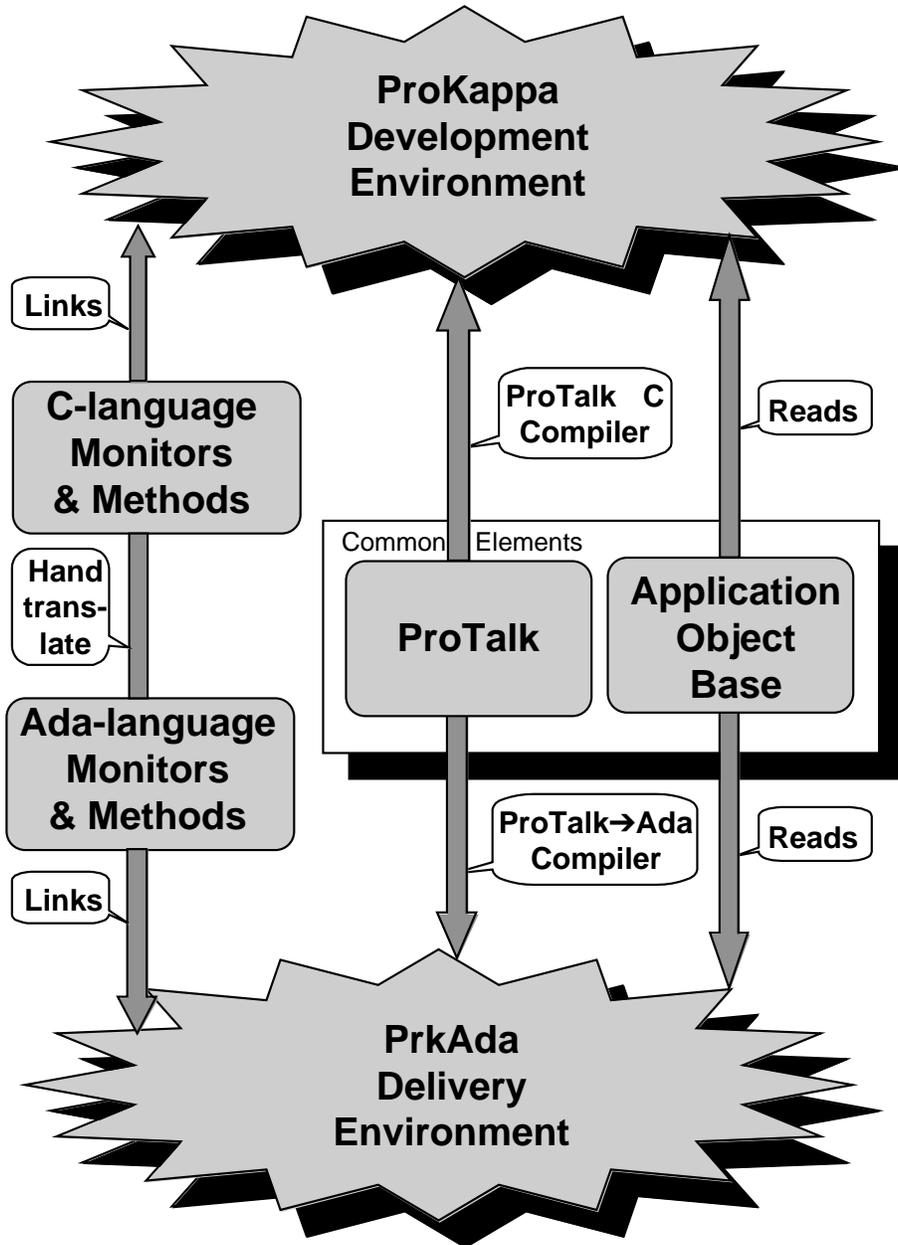


Figure 1. The development and delivery environments

6. A facility for reading and writing (the ASCII-style) of ProKappa object-bases.

The ProTalk-to-Ada compiler compiles a collection of ProTalk functions to Ada. ProTalk is a language that integrates the backtracking of languages such as Prolog [8] with conventional imperative constructs such as assignment, conditionals, and iteration. The ProTalk-to-Ada compiler is described in [5].

1.4. The PrkAda core

ProKappa implements a substrate in the spirit of Lisp and an object manager in the spirit and style of KEE. The underlying substrate includes a “variant-record” tagged datatype, with subtypes for symbols, cons cells, arrays, objects, several varieties of numbers, and so forth. The appropriate subtypes are garbage-collected. The object manager supports an object system much like KEE’s [1], including dynamic creation and deletion of slots and objects, dynamic rearrangement of object inheritance, slots with facets and demons on value-change, and object-oriented programming through function-valued slots.

The PrkAda core is a collection of over 20,000 lines of Ada code, divided into over 150 files. Almost all of these files are either the specification or body of an Ada package. A single covering package, Prk, defines the system datatypes and specifies approximately 350 user-level functions and procedures. These span almost all the functions in the development ProKappa environment.

The technical description of a fair-sized programming system is somewhat problematic. With the glow of authorship, we could clearly wax euphoric, down to the last semicolon, over any number of constructs. But in reality, the bulk of any such system is ordinary stuff. The PrkAda core, for example, has generic packages for list processing, routines for dynamically inheriting slot and facet values, mechanisms for invoking daemons on slot retrieval and modification, and specific subprograms for reading and printing PrkTypes. These subprograms are sometimes clever, but the cleverness is independent of AI and Ada—the same algorithms can be implemented in any sufficiently high-level language. Rather, the lessons worth discussing include:

- **Data polymorphism.** Ada is a strongly-typed language. Object-centered AI systems like KEE and ProKappa don’t require the declaration of the types of slot and facet values. Implementing dynamic typing in Ada requires defining a representation compatible with both the AI system requirements and Ada compiler restrictions. We use a variant-record mechanism.
- **Dynamic function binding.** AI systems allow functions (though not closures) to be slot values, while Ada strictly prohibits function-valued objects. Resolving this incompatibility while retaining modularity relies on a nested generic-case structure.
- **Object representations.** A delivery environment places different demands on its primitive data structures than a development environment. In particular, one wishes to sacrifice the developer’s ability to easily reorder the universe in return for greater run-

time efficiency. We describe the particular object representation used in PrkAda, and discuss the alternatives for delivery-system object implementations.

- **Automatic storage management.** AI programmers rely on garbage collection. No Ada compiler provides it. Garbage collection is difficult to do from the application programmer level: various protection mechanisms in Ada conspire to keep information needed for storage management from the application programmer. We describe an approach to garbage collection in Ada that combines a collection of storage management routines with a particular discipline (“hygiene”) of programming.

2. Data Polymorphism

An object system is fundamentally about storing and retrieving the values of slots. What can be the value of a slot? We'd like to store a variety of things: objects, symbols, various kinds of numbers, lists, arrays, temporals, strings, characters, and so forth. These values range from the small (characters, integers) to the large (objects, with about a dozen fields, many of which are themselves pointers). In a strongly typed language like Ada, only a single “type” of thing can be stored in any location. Type polymorphism is achieved in Ada with variant records. Let us call the kind of thing that can be the value of a slot a *box*. We have the choice of either making boxes be pointers to tagged records (Figure 2a) or themselves tagged objects that can contain either small, immediate values or a pointer to more complex data (Figure 2b). PrkAda uses this second alternative, thereby avoiding having to manage storage for small, common types (such as integers and booleans) at the cost of have a bigger boxes. (Systems such as C and assembler which allow pointer manipulation can embed the tag within the pointer structure, at the cost of a smaller integer space. This is common in implementations of Lisp and forbidden in Ada.)

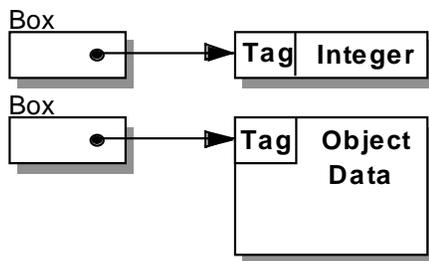


Figure 2a. Boxes as pointers

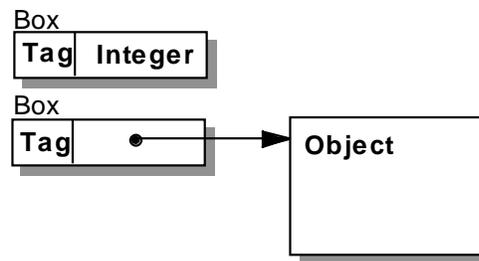


Figure 2b. Boxes as tagged data

3. Dynamic Function Binding

Dynamically bound subprograms appear in two places in ProKappa, either as *methods*, the target of message sending in object oriented programming, or as *monitors*, slot access daemon functions. The code for these dynamic subprograms can be created either by ex-

plicit programming or through the ProTalk-to-Ada compiler. The problem is that Ada does not allow pointers to functions. (A defect cured in Ada 95 [10].) While we could acquire function pointers through extra-Ada means (reverting to assembly language in an implementation-dependent fashion) such an approach would not be portable. Most straightforwardly, we need a kind of a box—a *method*—that can be assigned to a slot and used to invoke a function. Desirable goals for the method mechanism include:

- **Multiple collections of user functions.** We ought not expect the user to accumulate the code for all dynamic functions in a single place.
- **Independence of call structure.** One of the virtues of Ada is that package specifications carry enough information that users of the package need to be recompiled only when the specification changes, not on changes to the body (implementation). We don't want to have to recompile the entire system when the body of user packages change, and we want to have to recompile only trivial amounts of the system when user package specifications change. Thus, the interface between the system as a whole and the user method code must not introduce much in the way of compilation dependencies.
- **Appropriate method names.** User code and communications with the user should refer to user methods by user-chosen names, not by integers or generated symbols.
- **Efficiency.** We seek to minimize the computational effort to map between the box “token” that represents a function and the actual call.

To achieve the first three of these goals (at a small expense to the last) we employ the following mechanism:

- **User packages.** The user defines a number of packages, $P_1 \dots P_n$ that contain methods and monitor functions. In the specification of these packages, the user must supply (a) an enumerated type `MethodFn` and (b) a procedure “`Apply`” that takes a `MethodFn`, the arguments to be used for that function, and an answer result. (Ada has a built-in mechanism for mapping between enumerated type names and the strings that represent them.) In the bodies of these packages, the user must implement `apply` to call the appropriate user function for each method. (This `apply` is typically implemented with a case statement.)
- **GenMethods.** Generic package `GenMethods` takes an enumerated `MethodFn` type, an `apply` procedure of the above format, and an “ordinal” integer (indicating one of $P_1 \dots P_n$), and produces a number of useful utilities for that package, including routines for converting between the enumerated type and strings (and back again) and several `apply` methods for the package. These correspond to the various occasions where a method is called (e.g., as a by-needed daemon, as a procedure, and as an object-oriented programming method handler).
- **MethodFns.** The specification of package `MethodFns` provides the same functionality as `GenMethods`, without dependence on particular user method files. The body of

MethodFns instantiates the generic GenMethods for each user method package $P_1 \dots P_n$, and provides a case-statement format to select the appropriate user method package for each behavior. (Methods thus can be encoded as a pair of small integers, the first indexing the ordinal user method package to be used; the second, the element of that package's enumerated MethodFns datatype.)

Figure 3 illustrates the relationships among these packages.

When a user method package is recompiled without changing its specification, nothing in the rest of the system needs to be changed. When a user method package is modified to change its specification (for example, to add additional handlers), only the body of MethodFns needs recompilation. When the user adds a new user methods package, lines must be added to MethodFns to instantiate the generic GenMethods package and for an additional "when" clause in each of MethodFns's calling routines. In all cases, the specification of MethodFns does not change and the rest of the system is insulated from changes in the user code.

Lamb and Hilfinger [11] describe an alternative mechanism for dynamic function calling in Ada that employs tasks. Unfortunately, their mechanism does not support recursive procedures. Ada 95 fixes the lack dynamic function binding, providing an explicit mechanism for acquiring pointers to functions and invoking pointed-to functions.

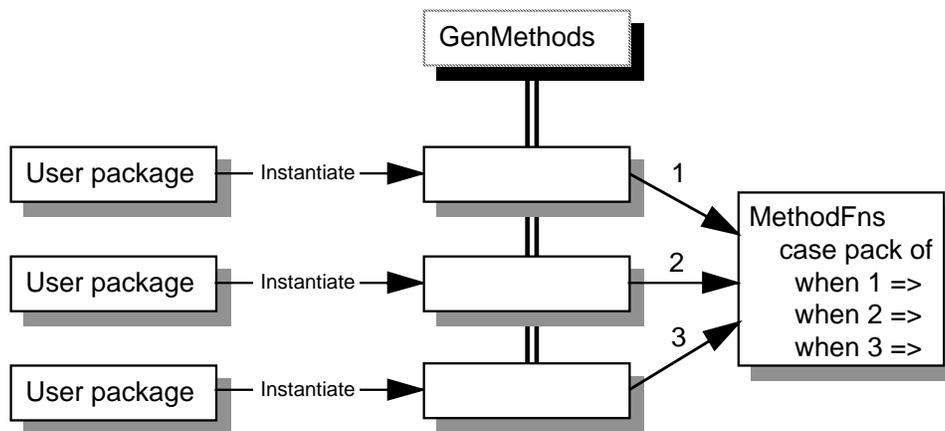


Figure 3. User package interface. Individual user packages are first instantiated into the GenMethods package. The resulting packages, each associated with a small integer, are then used in the MethodFns package. This mechanism allows the user to change both the specification and the body of user method packages without significant recompilation.

4. Object Representation

The cost of mapping from the symbolic name of a slot to its implementation dominates performance of a dynamic object system. PrkAda uses a dictionary mechanism to store such mappings.

In PrkAda, a dictionary is (primarily) an array of dictionary elements. Each element contains a slot signature: the slot name, its inheritance role and its slot type and flags. Each row of the array corresponds to a slot. Thus, a dictionary, D , whose third row has a dictionary element with name “Color” corresponds to a slot of name Color in position 3. Dictionaries also include a bit of auxiliary data—in particular, a hash value over the dictionary as a whole.

An object represents its slot information in two places: by which dictionary it uses and in a table of slot actual values. Each row in the slot table has space for a local value (values asserted directly for the object/slot), a combined value (the result of combining the object/slot’s local values with the combined values of its parents in that slot, governed by the particular inheritance role of the slot) and a list of that slot’s facets. The corresponding row in the dictionary defines the slot’s signature.

PrkAda was designed to be a delivery environment. We have found that for delivered systems, slots of an object are usually known at object creation, and it’s rare to change them. (In practice, modifying an object’s slots and parentage easily and dynamically is an important ingredient of a good development environment but is rarely needed in completed, delivered applications. Virtually the only subsystems that rely on dynamic objects at application execution are certain kinds of generic libraries, for example, generic database access tools.) The implementation of dictionaries reflects this assumption. The dictionary mechanism parses a number of possible slot descriptions and combines them with the descriptions of the dictionaries of the to-be-created-object’s parents. This gives a listing of which slots, (with which inheritance roles, slot types and flags) are to be in this object. This combination mechanism also checks for consistency between parent slot specifications. The system then searches for another identical dictionary. (Hashing is used for efficiency.) If such a dictionary is found, it becomes the dictionary of the new object. If not, a new dictionary is created and entered into the dictionary hash structure. Classes keep a cached dictionary to be used for single-parent child instances. This accelerates single-parent instance-object creation, by far the most common kind of object creation.

In searching for a particular (symbolically named) slot in an object, one needs to search that object’s dictionary for a slot of the given name. This is usually done by making dictionaries be hash tables [12]. In PrkAda, we chose to keep, for each symbol, an array of those locations taken by slots with that name in various dictionaries. The dictionary creation routines seek to optimize slot placement so as to minimize the length of these arrays and to reduce the search required within these arrays on slot lookup. This organiza-

tion has the potential of being a key optimization, as in certain cases we may be able to establish the slot index of a particular symbol at compile time.

PrkAda stores facets in association lists on object slots, sharing slot structures between parents and children whenever possible. This is less time efficient than the slot organization (and a bit less space efficient) because in ProKappa, facets can be created individually on slots on instance objects. (Slots must always be inherited, implying a bit of uniformity to the children of classes.) Figure 4 illustrates object representation.

Ada95 includes a tagged, extensible type system whose goal is to acquire the flexibility of object-oriented programming without sacrificing too much of the ability to capture type errors at compile time. This provides a single parent, static object system—a more straightforward foundation for building a variety of implementations of dynamic multi-parent objects.

5. Automatic Storage Management

The ProKappa substrate provides automatic storage management (garbage collection). Garbage collection relies on determining the closure of the “points to” relation over the “live roots” of a system. These live roots are (1) global variables and structures and (2) variables and structures allocated on the stack (the locals and parameters of the currently active procedures, back to the main calling program).

There are two major varieties of garbage collection algorithms, reference counting and mark and sweep. Reference counting keeps track of the number of active pointers to each cell. These reference counts must be updated every time a pointer is modified. A cell whose reference count goes to zero is garbage, and may be added to the free list. (Heuristically, a cell whose reference count reaches some maximum value becomes immortal.) Reference counting requires capturing every pointer modification in the storage management system, has the advantage of spreading out the effort of garbage collection throughout a program's execution, and the disadvantages of requiring space in each cell for the reference count and of being unable to collect circular structures that are nevertheless garbage.

In Ada, we lack access to the stack. To do mark-and-sweep garbage collection, we would need either to cheat (with some assembly language routine, thereby losing portability) or to avoid putting PrkAda pointers on the stack, keeping our own auxiliary stack instead (thereby considerably reducing the efficiency of our system and complicating program presentation and semantics—similar to the approach taken in [13]). In PrkAda, we implement a garbage collection mechanism that allows user code access to collectable items. Our storage management system uses reference counts, but similar technology could be applied to mark-and-sweep. Thus, garbage collection within the above restrictions requires a bit of user help—what we call *hygienic* behavior.

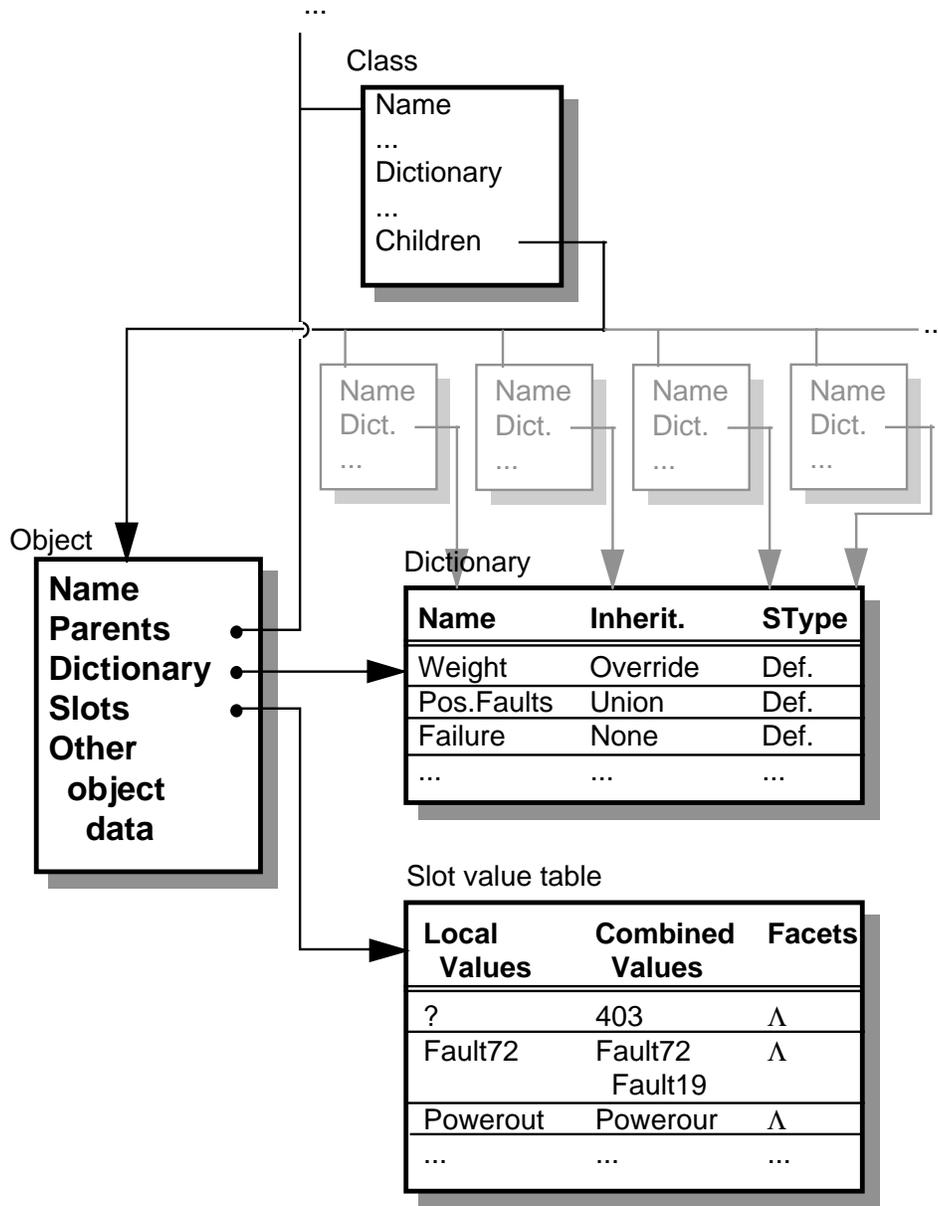


Figure 4. The anatomy of objects. One of the object's parents and several objects that share the same dictionary are also shown. For simplicity's sake, tagged-box objects have been omitted.

To do reference counts, one needs to be able to control the initialization, assignment, and finalization of each referable variable. (Finalization is the action taken when a variable goes out of scope.) Interestingly enough, Ada's limited private types do half of this—initialization and assignment of user-defined variables can be restricted to take place in private routines and can thus be captured by the storage management code. However, limited private types lack finalization, and functions with values in the reference domain created uncontrolled temporaries.

To clarify the above, it may help to provide a series of concrete examples. Consider a procedure *S*, where *box* is the type of item to be reference-counted:

```
procedure s (x : box) is
  z : box;
begin
  z := x;          -- <1>
  p (z);
end s;             -- <2>
```

At step <1>, the cell represented by *x* should have its reference count incremented. At <2>, when *z* goes out of scope, it should have its reference count decremented.

In Ada, by making *box* a limited private type we can “capture” events such as <1>. Thus, if the routines

```
procedure initialize (var : in out box; val : box) is
begin
  adjust_reference_count (val, +1);
  var := val;
end initialize;

procedure assign (var : in out box; val : box) is
begin
  if is_bound (var) then
    adjust_reference_count (var, -1);
  end if;
  initialize (var, val);
end assign;
```

are used instead of the assignment (*:=*) operator, we can capture events such as <1>. (By making boxes be a limited private type, we preclude the use of the *:=* and compel the programmer to call our *assign* procedure.) However, Ada has no similar restriction for references that go out of scope. It must become part of our hygiene to clear these variables at the end of their context.

```

procedure clear (var : in out box) is
begin
    adjust_reference_count (var, -1);
end clear;

```

We do not need to actually assign anything into var, as it is about to go away, anyway. Our hygienic approach to the program is then

```

procedure s (x : box) is
    z : box;
begin
    initialize (z, x);
    p (z);
    clear (z);
end s;

```

A slightly more hygienic approach would be to have an unwind-protect to insure that z is cleared even if an exception is raised:

```

procedure s (x : box) is
    z : box;
begin
    initialize (z, x);

    begin
        p (z);
    exception
        when others => clear (z); raise;
    end;

    clear (z);
end s;

```

This gets us the first rule of memory-management hygiene:

(1) Clear local variables on program exit.

Let us next consider the call to procedure p:

```

procedure s (x : box) is
    z : box;
    procedure p (y : box) is
    begin
        ... -- <3>
    end p;

```

```

begin
  initialize (z, x);
  p (z);
  clear (z);
end s;

```

Do we have to do anything at <3>, inside p, to ensure the reference to y? Probably not. As long as z in calling routine is not modified by p or anything p calls, it retains a reference to its cell. In that situation, p does not have to do anything to guarantee the reference. This gets us to the second rule of good hygiene:

(2) Parameters to subprograms do not require reference count maintenance, as long as a reference to the parameter will not be modified while the procedure is executing.

That is, if you are sure that something keeps a pointer to a parameter during the subprogram call, the called subprogram does not have to perform any reference count maintenance of its own. Since this desirable state of affairs usually holds, we put the burden of keeping parameters pointed-to on calling routines. Typical ways that a calling routine can be sure that its actual parameters are safely pointed-to during a subprogram call include:

- Making the actual parameter be something with a known global reference. That is, being sure that something else permanent also points to the parameter (e.g., an item on the answer stack, below, or a global constant).
- Making the actual parameter be a local variable of the calling subprogram that is not visible to any other subprogram. Thus, no called procedure can modify the parameter value; it remains constant for the duration of the call.
- Making the actual parameter be a local variable that, while visible in other subprograms, is clearly not modified by any of them. Locals visible to other subprograms are declared within this subprogram; the variables they modify are visible in those subprograms' text.
- Inductively, by using a formal parameter of the calling subprogram.

A critical point in this algorithm is protecting global variables: structures visible throughout the program that may be modified anywhere. In PrkAda, we ensure parameter safety by encapsulating the reference to global structures in a purely functional interface. The answer stack mechanism, described below, thereby provides the necessary additional, durable reference.

However, one problem persists. Consider the function:

```

function fun1 return box is
  a : cons_cell;
  procedure cleanup is
  begin
    clear (a);
  end cleanup;

begin
  initialize (a, cons (nil, nil));
  cleanup;
  return a;
end fun1;

```

This function won't work. The problem is that we've disposed of the cons cell `a` before returning it. Similarly, if we had omitted the cleanup operation on `a`, then, after the value returned from `fun1` had been processed, it would have an additional, spurious reference to `a`. This would be unrecoverable garbage. We have to keep the return value "long enough" for it to be processed by the caller, but not forever. How long is "long enough"? It can be quite long, as illustrated by the expression

$$f(g_1(x_1), g_2(h_{21}(x_{21}), h_{22}(i_{221}(x_{221}), i_{222}(x_{222}))))$$

While computing $i_{222}(x_{222})$, the temporary results of i_{221} , h_{21} , g_2 , and g_1 must be preserved. Clearly, a single cell for answers is insufficient; a stack is needed. (We need one such stack for each concurrent thread.) This stack would serve to preserve a reference to the answer. Any intermediate results computed by, say, i_{221} , would also go on this stack. At any semicolon in a subprogram, items on the stack can be cleared back to the the stack top at the entry of that subprogram. Figure 5 shows the state of the answer stack while executing i_{222} . Our implementation includes several helper functions for storing a function's result on the answer stack. (A supplemental paper [7] discusses these in greater detail.) The most frequently used of these is `lreturning`, which clears the stack to the entry point and returns its given value, and `freturning`, which returns a value from a function call.

The appropriate hygiene for using this mechanism is to remember the stack pointer on function entry (an `anspoint`) and either (1) to store the answer in a particular variable, and return the value through the returning function, or (2) to use the `freturning` form to return the value of a function call. That is, a hygienic version of `fun1`, in style (1), is

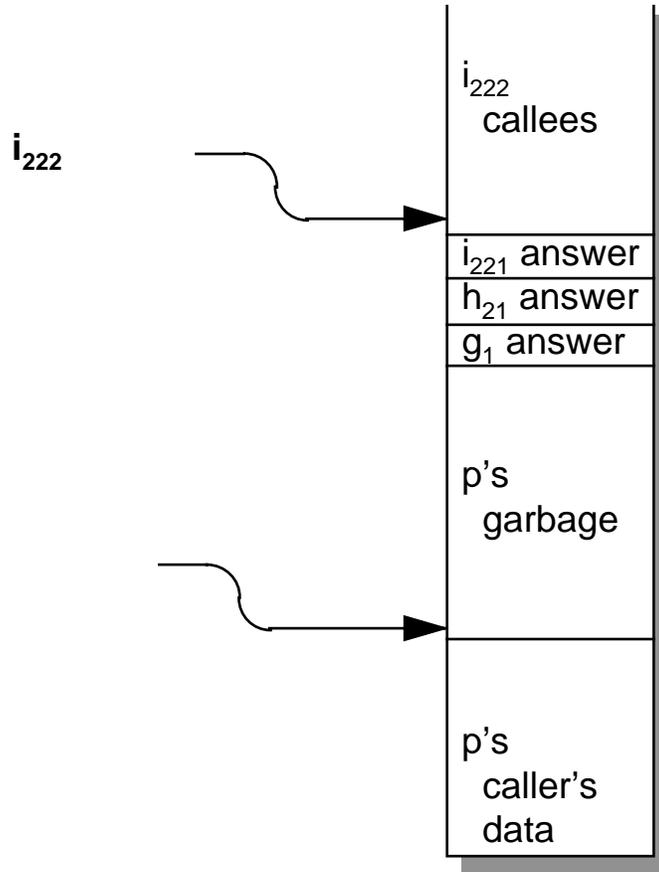


Figure 5. The answer stack. This figure shows the state of the answer stack while evaluating i_{222} .

```
function fun1 return box is
  ansp : anspoint := note_return;
  a : box;
begin
  initialize (a, cons (nil, nil));
  return lreturning (a, ansp);
end fun1;
```

Thus, the third rule of good hygiene is

(3) Use the answer stack for functional results; clear it as necessary.

Only functions (not procedures) need to use the answer stack. If the current return point is global (i.e., if the answer stack is implemented properly with an abstract data type), then any subprogram can call `clearstack` “at any semicolon.”

5.1. Alternative approaches

Baker [14] approaches these problems by defining a generic mechanism for the creation of variables (nesting generics). Nesting generics ensures control over all three steps crucial to reference counts. This approach has, however, several disadvantages: (1) Virtually all programs become embedded in generic handlers, resulting (for almost all compilers) in considerable code expansion, (2) Nesting generics is unable to handle pointers that are components of structures or arrays, (3) the uniform use of exception catchers makes debugging exceptions more difficult, (4) nesting generics does not solve the function problem, and (5) we've had too much experience with certified Ada compilers that could not properly compile generics to rely on their complex use.

The approach we described has some flavor of attempting to get the actions of the nesting generics approach without the use of the generic mechanism. We also provide a solution to the problem of functions returning collectable boxes with the answer stack mechanism.

Ada 95 provides *controlled types* with user-controlled initialization after creation, finalization before destruction and adjustment after assignment. Using controlled objects, “good hygiene” can be enforced by the compiler.

6. Object and Slot Organization

The particular data structures used in PrkAda were a choice among many possibilities. It is worthwhile explicating the engineering choices, the environmental assumptions behind those choices, and the alternatives. If we may oversimplify, there are eleven primary activities of a ProKappa-like core. Table 1 lists them in order of frequency of occurrence in running applications (based on our experience with a variety of applications). Note that with the dynamic, instance-method model of object oriented programming, sending a message to an object is effectively doing a slot retrieval followed by an apply.

In frequency, slot value retrieval dominates the others, and slot value modification dominates the remaining activities. Unlike languages like C++, the slots of an object can dynamically change. Slot access time is mediated by the time it takes to find the storage of the slot within the object. The art of KBS tool construction thus hinges on minimizing this time without paying either too heavy a penalty in space or unnecessarily restricting the variety of operations available to the user. From an engineering point of view, the design issue for such an object system turns on the space and time efficiency of these operations. Clearly, there are some engineering tradeoffs between these tasks. For example,

Table 1: Primary activities of a dynamic object-system core

1. Retrieving the values of slots.
2. Storing values into slots.
3. Retrieving facet values.
4. Storing facet values.
5. Running monitors on slot access or modification.
6. Creating objects.
7. Causing slots to exist at the object level, either directly or by inheritance.
8. Causing facets to exist at the slot level, either directly or by inheritance.
9. Inheriting slot values from parents to children.
10. Inheriting facet values from parents to children.
11. Deleting objects

- Performing inheritance when a class value changes slows storing values into class slots, but speeds retrieving values from slots (alternatively, one could perform inheritance at retrieval time, if storage to classes was more common than retrieval from instances or disallow dynamic inheritance, producing a more “restricted” system);
- Storing slot structures as an association list speeds slot creation but slows slot access;
- Keeping monitor information in a fixed place in the slot structure speeds finding out if an object has a monitor and what that monitor is, but requires extra space for the monitor data structure on each slot (whether or not it has attached monitors) and complicates the overall system with additional code to handle monitor inheritance as a special case.

The appropriate tradeoffs are a function of the presumed use pattern of the system. The usage pattern of table 1 suggests that the mapping from slot names to value addresses (and related information) be done by a fast yet less space efficient action like hashing, while simpler structures, such as property lists, be used for facet structures. The approach taken in an earlier version of the system was to provide a single, universal hash table for a specific slot structure. The structure varied by whether the slot is a class slot or an instance, and by the inheritance role of the slot. This universal hash table expands, tree-like, into other hash tables when the density of any particular bucket is too large.

This representation proved to be space efficient but not as time efficient as desired. For the current version of the system, we used the dictionary mechanism described above. We associated slot information of each object in two structures: a dictionary to store “slot constant” data such as the mapping between slot names and locations, slot types and flags, and a value structure to store combined and local values and facets. Objects that have the same slot signature share dictionaries; objects have their own local value structure. The current dictionary mechanism relies on a directed sequential search of the possible positions in a dictionary for a given slot. For example, a symbol, say, color, might have as its slot list (2, 5, 1). To find the color of an object, we look in that object’s dictionary in successively, the second, fifth and then first rows, checking each to see if it defines color; if none of those rows match, the object does not have a color slot. (The particular engineer-

ing assumption here is that there are few possible locations of any given name. Under that assumption, simple sequential search is efficient than complex search algorithms. Self-organizing lists [15, §6.1] could be used to rearrange the order of a slot name's position list.) If the user has been parsimonious in choosing slot names, or the application compiler has been clever about placing names in common locations, this algorithm is likely to find the desired slot quickly. (Of course, we could have made the dictionaries into hashing structures themselves.) Relying on more global perspectives of slot usage, and leaving no expansion space in objects for dynamically defined slots serves to speed up slot value retrieval and storage at the expense of slot creation.

6.1. What can be optimized

ProKappa, in its full generality, allows the dynamic creation and modification of object structures. Such facilities have an accompanying cost—to the extent that data structures require additional generality, the routines for manipulating those structures can take longer, and, to a lesser extent, the code for that manipulation requires space. PrkAda is intended as a delivery system, one where the structure of the system can be examined by a compiler. We work under the assumption that we can consider the entire program and knowledge base in optimizing the system behavior. Such optimization requires inferences on the part of the compilation system about the code. A worthwhile extension of this work would be to allow the user to provide such inferences (as pragmas) directly.

What optimizations are possible? We note that the most efficient slot representation is an array access, where the calling program knows (has as an integer constant) which element of the array stores the value for the specified slot. (Object systems such as C++ accomplish this constant-time access by restricting their message handlers to manipulating a single class of unchanging object. AI-style object systems access slots by name through a collection of slot accessing and modification functions. In that architecture, the slot name can be dynamically computed. The least efficient slot representations require the full generality of property lists or hash tables that can gain and lose elements.)

Starting from this full generality, let us consider possible restrictions on system use and the inferences we can make about the resulting data structures.³

- 1. No dynamic slots.** No slots can be created or deleted by the running program. The slot data structure can be a fixed size, and does not need to be dynamically reallocated.
- 2. Most slots are not dynamic.** That is, most slots can be guaranteed to exist for the entire program execution. The slot data structure can be a fixed size, with provision

³ This list represents the more interesting restrictions we considered in actually developing the system. Of course, other optimizations are possible, and particular implementations lend themselves to some optimizations that are inappropriate for others.

for a “less efficient” structure to store slots that are dynamically created during program execution.

3. **A specific slot is not dynamic.** That is, objects in a particular class are guaranteed not to lose this slot. A compilation system could hardwire access to such slots when the slot name is provided as a constant, at the cost of complicating the slot access code in general (as it must now deal with an additional special case).
4. **Single-site slot introduction.** A specific slot name is introduced in only a single (or controlled) set of places in the object hierarchy. It may be possible to speed access to the location of that slot, avoiding hashing through the dictionary to find out where it is. If done generally enough, it may be possible to avoid most searches of object dictionaries.
5. **Fixed slot types.** Slots do not change slot type after creation. Thus, the value table of the particular slot does not change. (In our style of object system, changing the slot type changes which objects inherit the slot.) Efficiencies can be gained by grouping all slots of a given type together.
6. **Fixed slot inheritance roles.** Slots do not change inheritance role after creation. (The inheritance role determines how parent and child values are combined to create the visible child value.) This saves the code required to recompute all values in a hierarchy.
7. **Slots are never deleted.** This saves checking for deleted slots and simplifies algorithms by not requiring reclaiming space for deleted slots.
8. **A class has no facets.** This saves the space for the facet lists for such objects, at the cost of introducing another check in the facet manipulation code.
9. **A class has only inherited facets.** If slots are implemented with a single local facet list (implying run-time search of their parents) we can save the space for such a list. If parent facets are combined with their child facet lists at inheritance time, having no local facets makes this combination particularly straightforward for single-parent objects.
10. **A class has no inherited facets.** Similarly, this saves the local facet list for such objects, and the concurrent search.
11. **A particular slot has specific facets.** We may be able to allocate a specific structure for these facets. The most obvious application of this optimization is for facets with specific system meaning, such as monitors or type-checkers.
12. **Objects do not change parents.** This saves the code for rearranging the slots of an object dynamically, and insures that if the slot index of an object/slot has been computed, it need not be recomputed.
13. **Objects have only a single parent.** This saves the code for more complex inheritance algorithms, and allows more sharing between child and parent structures.

- 14. Objects are never deleted.** We do not have to include code to check for deleted objects, nor code to actually mark an object as deleted and reclaim as much of its storage as possible.
- 15. A particular call refers to a constant slot name.** We may be able to compile this to a fixed index.
- 16. A particular call refers to a constant facet name.** We may be able to search more directly for this facet, or even to compile it to a fixed index.
- 17. A particular call refers to a constant object.** We may be able to tie this object to a particular place in the object space.
- 18. No dynamic classes.** If class objects cannot be dynamically created, then certain global optimizations may be possible. For example, it may be possible to assign all uses of slot “color” to the first element in every object’s slot table, allowing compilation of retrievals and changes of color to be constant time.
- 19. No dynamic objects.** Objects may not be dynamically created. If objects cannot be dynamically created, then we can pre-determine where all objects go, saving the dynamic mapping between object names and their data structures.

Most of these restrictions are eminently reasonable, particularly for a delivery system. In practice, running systems rarely change a slot's type or inheritance role. Others are more application dependent. In PrkAda, we chose to include restrictions 7, 11, 12, 15, and 16; for small additional quanta of effort, restrictions 7, 11 and 12 could have been removed. Additionally, the implementation was constructed with assumptions 2 and 3 in mind, and is thus less efficient when these restrictions are violated.

7. Concluding Remarks

PrkAda illustrates the possibility (and difficulties) of developing a delivery environment for AI applications in Ada. On one hand, Ada restrictions such as the lack of a language-based garbage collector and functional values considerably complicate the program development process. On the other hand, given the compilation machinery we have developed, it has proven relatively straightforward to port our demonstration applications to a machine-independent, portable Ada. The morals, perhaps, are (1) it's possible to do AI in Ada, though not particularly easy or fun⁴, and (2) by understanding the restrictions of a

⁴ The authors must also confess, inveterate Lisp hackers that they are, that the experience of working with Ada has given them considerably greater respect for aspects of programming emphasized by Ada. A major goal of Ada design was maintainability. This is reflected in both the greater pains that the programmer must go through in creating a system in the first place, and the fact that, verily, the resulting code is more easily maintained.

run-time environment, it is possible to compile more efficient systems than those required by general development environments.

PrkAda has been used for various demonstration projects within IntelliCorp [7] and has been applied by a customer to the development of aircraft engine maintenance software.

Acknowledgments

This work was performed while the authors were working for IntelliCorp, Inc., and was supported by NASA/Marshall Space Flight Center under contract NAS 8-38488.

References

- [1] R. E. Fikes and T. Kehler, The role of frame-based representation in reasoning, *CACM*, **28** (1985) 904—920.
- [2] C. Williams, *ART The Advanced Reasoning Tool—Conceptual Overview*, Inference Corp., Los Angeles (1984).
- [3] Carnegie Group Inc., *KnowledgeCraft User's Manual*, Pittsburgh, Penn. (1991).
- [4] IntelliCorp, Inc., *ProKappa Reference Manuals*, Pub. No. PK2.0—RM1—2, (1991).
- [5] R. E. Filman and P. H. Morris, *Compiling Knowledge-Based Systems to Ada: The PrkAda ProTalk Compiler*, *International Journal on Artificial Intelligence Tools*, this issue, (1997).
- [6] R. E. Filman and R. D. Feldman, *Annual Report: Compiling Knowledge-Based Systems Specified in KEE to Ada*. IntelliCorp, Inc., Mountain View, California (1991).
- [7] R. E. Filman and P. H. Morris, *Implementation Notes for PrkAda*, <http://www.best.com/~morris/ftp/prkimp>, (1997).
- [8] R. Kowalski, *Logic for Problem Solving*, North-Holland, New York, (1979).
- [9] D. H. D. Warren, *An Abstract Prolog Instruction Set*, Technical report number 309, Artificial Intelligence Center, SRI International (1983).
- [10] Ada 95 Mapping/Revision Team, *Programming Language Ada: Language and Standard Libraries*, ISO/IEC DIS 8652, Intermetrics, Cambridge, Massachusetts, (1994).
- [11] D. A. Lamb and P. N. Hilfinger, *Simulation of Procedure Variables Using Ada Tasks*, *IEEE Trans. on Soft. Eng.* **VSE9** (1983) 13—15.
- [12] R. E. Filman, *Retrofitting Objects*, ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA—87), Orlando, Florida (October 1987) 342—353.
- [13] M. Yen, *Using a dynamic memory management package to facilitate building Lisp-like data structures in Ada*, *Proc. AIDA-90* (1990) 85—93.
- [14] H. Baker, *Structured programming with limited private types in Ada: Nesting is for the soaring eagles*, *Ada Letters* **11** (1991) 79—90.
- [15] D. Knuth, *The Art of Computer Programming: Volume 3, Sorting and Searching*, Addison-Wesley, Reading Massachusetts (1973).