Implementation of an Ascend Interpreter

by

Thomas G. Epperly

EDRC 05-29-89

Implementation of an Ascend Interpreter

Thomas G. Epperiy

Engineering Design Research Center

Table of Contents

List of Figures

1. Introduction

Implementing an interpreter for the Ascend modeling language has played an important role in the formal specification of the language itself. This implementation has been written primarily as a research tool and not a commercial product. The principals which guided the developement of the interpreter are presented in approximate order of importance:

1. correctness	The primary goal has been to write an interpreter which accurately matches the language description. The interpreter must reliably execute as many of the language functions as possible. [3]
2. flexibility	Since the definition of the language was modified while the interpreter was being written, the interpreter had to be able to adapt to new language constructs or changes in the language definition. Two early attempts were scrapped because they lacked the required flexibility.
3. speed	The code has been optimized for speed whenever possible. However, considerations of running speed never influenced the language definition; and the correctness goal always took precedence over speed.
4. generality	Special case processing as a means of improving performance has not been exploited. Special cases may arise from the semantics of the current language definition, or a notion of typical usage of a certain construct. An example of a special case is treating boolean instances named "fixed" somehow differently from boolean instances with other names. It was felt that embedding special cases in the interpreter code would compromise the flexibility and adaptability of the interpreter.

The goals of a commercial package differ from these ones in an orientation towards portability, speed and generality. A commercial system would need to be easily portable to all the major workstations and perhaps even mainframes. Speed would have to be more of a priority; perhaps even to the point that it would dictate some changes in the language. Finally, the interpreter might need some special purpose routines to handle customers' special uses.

The goal of this research is to produce an incremental interpreter; however, it was decided that the initial implementation should be batch-oriented. The primary reasons for making this decision were, the difficulty of writing an incremental interpreter, and the continual changes in the language definition The batch interpreter has been adequate for demonstrating and testing most aspects of the Ascend language

2. Data Structures

Getyourdatastructurescorrectfirst,andtherestoftheprogramwillwriteitself.

DavidJones
Assen,TheNetherlands[1]

This part of the paper will attempt to describe the data structures that were chosen for the Ascend interpreter. This section starts out by looking at the data structures at a very low level of detail and gets increasing more detailed as the paper progresses. In order to fully understand the data structures it is essential that the reader study the figures well; words alone cannot completely explain these structures. The symbols defined in these figures are used consistently throughout the whole paper. Please note that this paper will not cover the details of all the types used, but it will describe all the important or difficult data structures.

2.1 Type library

When the file containing the Ascend code is parsed, a type library is created. This library is just a machine representation of the definitions found in the file. The type library must know the ancestry of each type and be able to look up each definition in the ancestry. Each library entry must have statements and initialization code (atoms require dimensions and default in addition) which constitute the type's definition.

Another possibility would have been to create an instance of each type found in the file and store the instance in a type library. In this situation, it might have only created a *partial instance* because the value of the integer and string instances used in the type's definition were not known. (See Section 5, page 18 for an explanation.) Then, to create an instance of a type, the instance in the type library would have been copied. This idea was rejected because processing the file would take too long, and because it would create instances of types that would never be used. This idea would be more practical for library definitions which could be pre-processed.

All of the types in the library are globally declared; there are no context dependent types or hidden types. This suggests that the types should also be stored in a table which does not have any hierarchy associated with it. The other possibility would be to store types in the hierarchy given by the ancestry tree. Because accessing type is primarily done without regard to ancestry, this implementation uses one

[1]Jon Bentley, Communications of the ACM, September 1985, Vol. 29, Number 9, Programming Pearls, "Bumper-Sticker Computer Science"

table to store the type library. The ancestry information is maintained by including the type's parent in the type's library entry. This choice makes it easy to retrace a type's inheritance, but makes it difficult to find the children of a type. To find the children the whole type library must be scanned looking for types whose parent is the type of interest. The instantiation code does not need to know the children of a type. An interface is the only client code that would need to know a type's children.

Because types are looked up by name, a hash table was chosen to store the type library. Each bucket of the hash table contains a list to avoid problems with collisions. The hash table gives rapid lookup based on name. Hash tables have better average case running times than lists or sorted lists. (See Section 3.2 for details on the hash table and Section 3.1 for details on the list.) Figure 2-1 shows a graphic representation of the type library. In this simplified representation, only the "real" and "generic_real[M] entries are shown, which are only some of the entries contained in the two buckets. The rest of the library entries are not shown.

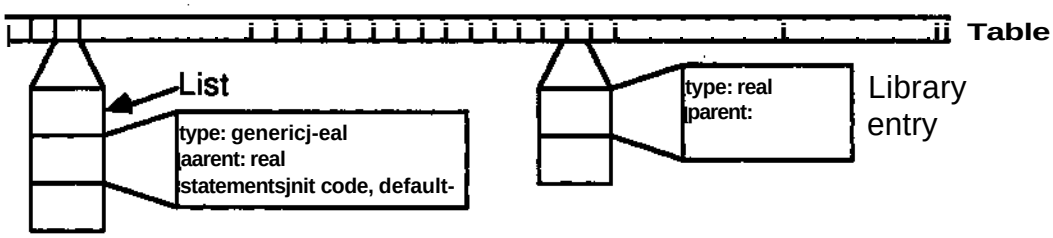

Figure 2-1: Global Type Library

2.2 Instances

2.2.1 The Instance Tree

Ascend is an object-oriented language, and the objects that Ascend code produce and manipulate are instances. The choice of the data structure for instances was motivated by the fact that an instance is itself built up from other instances; establishing a hierarchical parent-child relationship. It is important to structure the information in a hierarchical manner because this hierarchical structuring puts the information in context. For example in a system of 3000 equations and 3300 variables, it would be advantageous to break them up into meaningful groups. A tree was the data structure chosen to represent instances. In this report, the expression "instance tree" refers to an instance and its children. and "instance" refers to an instance without its children. Figure 2-2 shows a piece of an instance tree. (Clouded areas indicate parts left out.) The links shown here are parent-child links and are different than

those discussed in the language definition. Notice that an instance can have two parents as a result of an ARE_THE_SAME (i.e. .values[5][toluene] and .calc_model.component ARE_THE_SAME). The name of an instance is stored in the link rather than with the instance, because an instance can have multiple parents which each use a different name to refer to the child instance.

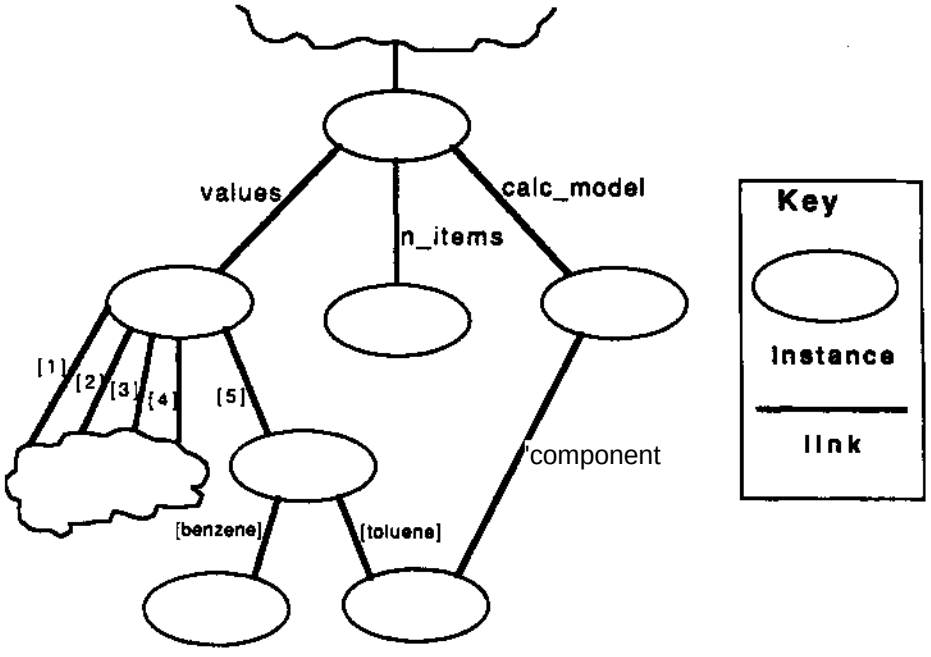

Figure 2-2: Conceptual Instance Tree Data Structure

2.2.2 Parent-Child Links

Between every parent and child in the instance tree there is a link data structure. The link contains the child's name in the parent's context and pointers to both the parent and the child. The instance tree has to be linked both ways (from parent to child and child to parent) because when an ARE_THE_SAME is executed, the parent to child links must be changed and the interface code needs to be able to determine the parents of an instance. The link differentiates between the three types of names that an instance can have: identifier, string index, and integer index. Figure 2-3 shows the link in more detail.

2.2.3 Representing ARE_THE_SAME

When instances have been declared ARE_THE_SAME, there are multiple links between the parents and child. Figure 2-4 shows an instance with two parents. Encoding ARE_THE_SAME this way fits the semantics of the operator, and the interpreter does not have to worry about enforcing the ARE_THE_SAME because the multiple instance trees have been merged to form one instance tree. Any

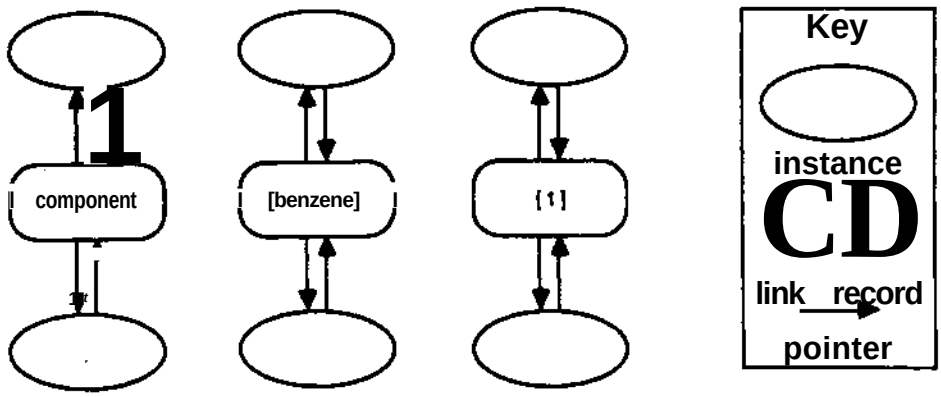

Figure 2-3: Detail of Parent-Child Link

further refinements automatically effect all the instance trees that were merged. This method of merging instances will not permit the undoing of ARE_THE_SAME operations because merged instances cannot be separated without a great deal of work.

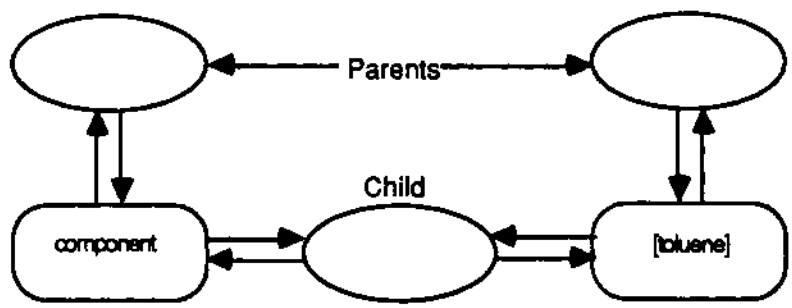

Figure 2-4: Detail of ARE__THE_SAME Links

2.2.4 Representing ARE_ALIKE

The ARE_ALIKE operator must be able to create and maintain cliques of instances who have the same type. Using a list of cliques, with each clique being a list of instances, was the first method considered. This was rejected due to the time required for the system to search through the lists. The method chosen instead uses a circular linked list to group the instances that are declared ARE_ALIKE. This circular linked list does not use the list module described below. Each instance has a clique pointer which points to an instance which is in the clique or NIL if the instance is not in a clique. Instances declared ARE_THE_SAME are inherently ARE_ALIKE because they are merged. Figures 2-5 and 2-6 illustrate this method.

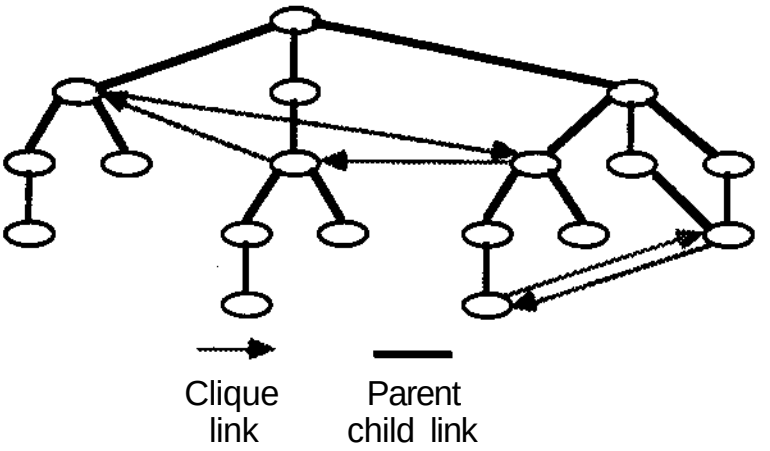

Figure 2-5: Instance Tree Indicating Cliques

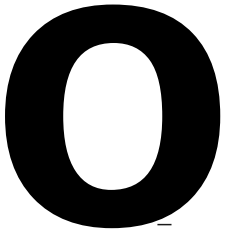

Figure 2-6: Circular Linked List Clique

2.2.5 Common Attributes of All Instances

All instances share the following attributes: parents, clique pointer, pending flag, base type, and information pointer. The parents attribute is a list of all the parents of the instance. This list allows children to point back to their parents in the instance tree (as described in Section 2.2.2 above). The clique pointer is used for the circular linked list (as described in Section 2.2.4 above). The pending flag is used during instantiation, and its importance is explained in Section 5.2.1. The clique pointer and pending flag have no significance for array and relation instances. The base type indicates what kind of instance the particular instance is. It can indicate any of the following:

- array
- relation
- model
- real
- integer
- boolean
- unit

Any of the last four are atoms. This attribute does not attempt to store the refined type. For example, instances generic_real or temperature would have base type real. The last attribute that all instances (including arrays and relations) have is the information pointer. What the information pointer points *to* is

indicated by the type of instance. In general it points to where the specific information of the instance type is stored; the details of each instance type are in the sections below. Figure 2-7 illustrates an example of an instance.

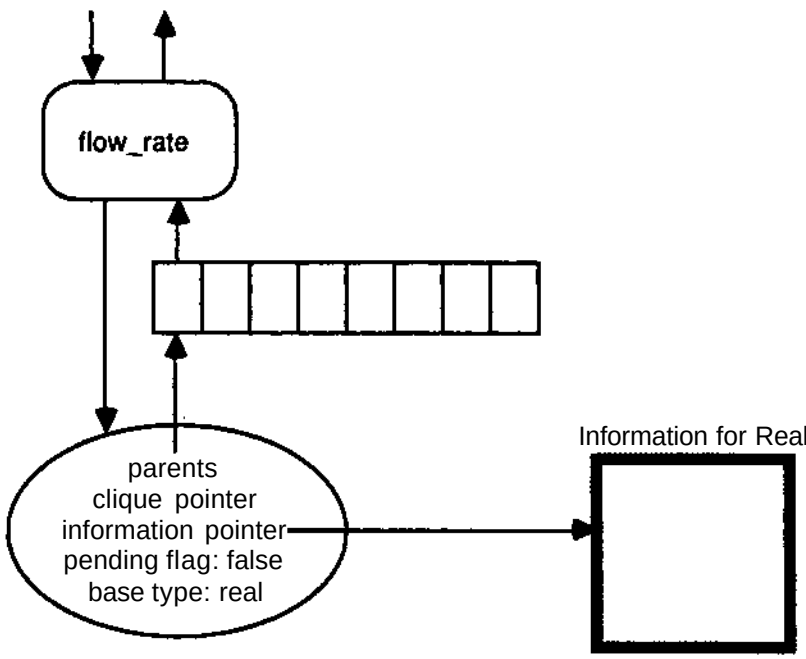

Figure 2-7: Command Instance Attributes

2.2.6 Arrays

In the language description, arrays are not treated like objects; they are treated as naming shortcuts. For example, you cannot declare arrays ARE_ALIKE or ARE_THE_SAME; however, arrays can be ARE_THE_SAME'd as the result of ARE_THE_SAME'ing models which contain arrays. In the implementation, arrays are considered to be instances and objects. This choice is more flexible than having arrays be naming shortcuts, because it allows both interpretations. This distinction becomes important when you consider arrays with multiple dimensions as shown below in Figure 2-8. It is important to note that in both cases a[1]['example[f]] is a real. Treating arrays as objects gives more hierarchy to the instance tree.

Arrays in Ascend are not like arrays in other typical algorithmic languages. The main differences are listed below.

1. Arrays can be indexed by strings.
2. The size (dimension) of arrays is not specified in the declaration.
3. The elements in an array are created through use. Referring to a[1] causes a[1] to exist.

Arrays are Objects	Arrays are Naming Structures
a[integer][string] IS_A real; a is an array indexed by integer of arrays indexed by string of reals a[1] is an array indexed by string of reals a[1]['example'] is a real	a[integer][string] IS_A real; a is an array indexed by integer and string of reals a[1] has no meaning a[1]['example'] is a real

Figure 2-8: Different Interpretations of Arrays

This has the unusual consequence that a[1] and a[3] can exist without a[2] existing.

4. Arrays can contain any defined object (models or atoms).
5. Elements in an array can be refined without having any effect on the other elements in the array. This implies that arrays can contain objects of different types; however, all the types must be a refinement of the array's base type.

These differences have had a large impact on the data structures and instantiation procedure of the Ascend interpreter. The effects on the instantiation are discussed later (See Section 5, page 18.) Unlike most language processors, the Ascend interpreter cannot store arrays in a block of contiguous memory and use the index to calculate the address of the array element. Instead, a data structure which can hold any number of items and can look up the child using integers or strings must be used. Probably the best solution to the problem of array element storage would have a chained hash table (like the one used for the type library). However, it would have needed a different hashing function, that would give rapid lookup for children. One disadvantage of the hash table was that it could not satisfy the interface's requirement for array elements in sorted order. This would have required the interface to sort the list each time it wanted to present the array elements. Because of this inflexibility it was decided to use a list instead of a hash table. This list is kept sorted by the index value which improves the average case lookup time as compared to an unsorted list and satisfies the needs of the interface. [2] Details of the list are discussed in Section 3.1, starting on page 15.

The information record of an array instance has a base type, index description, and array elements associated with it. The base type is the type that is stated in the array declaration. All elements in the array start out with that base type. Later they can be refined to be a different instance of a more refined type. The index description is a list which stores the index types in the order that they appear in the

[2] Runtime analysis indicates insignificant amounts of CPU time are spent looking up array elements. There is not enough justification at the current time to merit changing the hash table module.

declaration. For example, the declaration a[integer][integer][string] IS_A generic_real would create an array with the description list containing integer, integer, and string in that order. The array elements are stored in a list as discussed above. Figure 2-9 gives an example showing what the information pointer of an array instance points to.

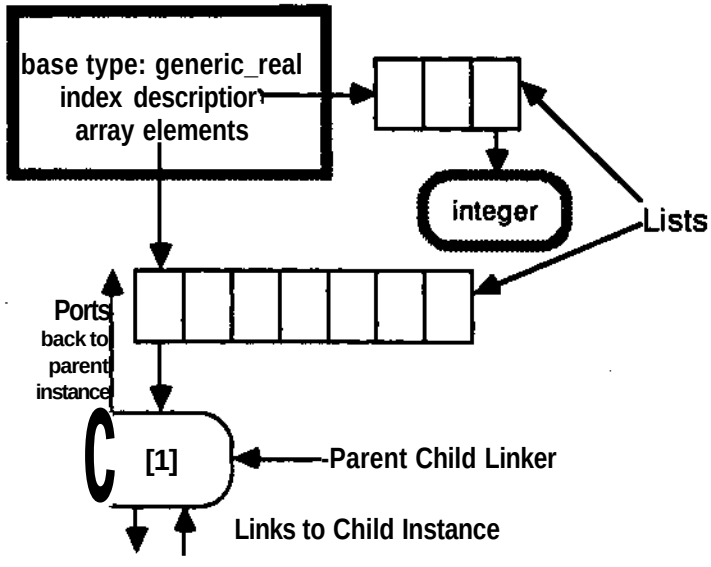

Figure 2-9: Illustration of Array Element Storage

2.2.7 Models

The information pointer for a model instance contains the type name, pending instruction list, initialization code, and children of the model. The type name is the instance type. The pending instruction list is a list of statements from the type definition that have not been interpreted yet. (See Section 5.2.1 for more details.) The initialization code is a copy of the definition code from the type definition. At the time of this paper, the initialization code consists of a list of assignments. The children of the model are stored in a chained hash table similar to that used in the type library. Because of its efficiency in this process, a hash table was used to give rapid lookup of the named children. The hash table size is kept small to prevent wasteful memory loss. By adjusting the size of the hash table, it is possible to trade faster lookup time for more memory. This structure is illustrated below in Figure 2-10.

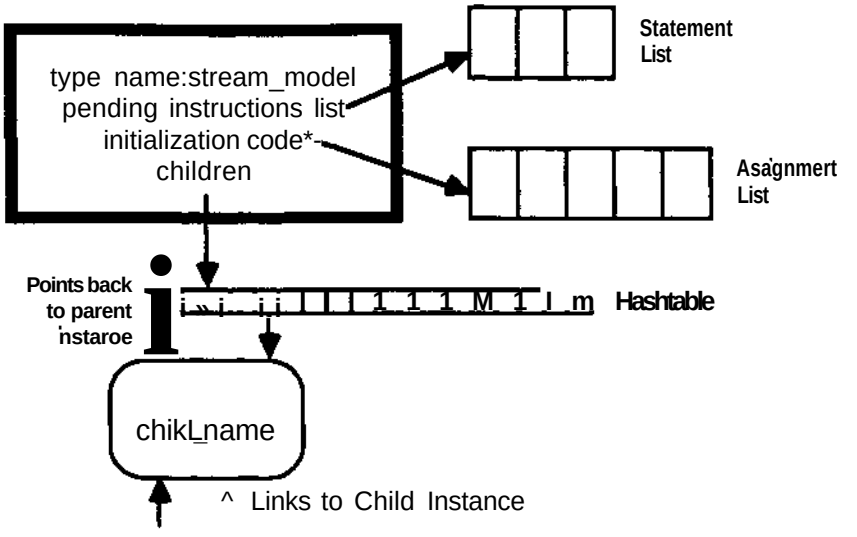

Figure 2-10: Details of a Model's Information Record

2.2.8 Base Types

2.2.8.1 Atoms in General

In the first version of the interpreter, there was no difference between an instance of a refined atom (i.e. generic_real) and a base type (real, string, boolean, and unit). Every atom used the same amount of memory because they all had the memory reserved for storing children, pending statements lists, initialization code, and the type name. Refined atoms needed all this information, but base type atoms did not use any of it. In test cases about half of the instances in an instance tree are base types, coming mostly from the fields of generic_real. Because of these two reasons, base types were made into a special case of atoms.

The difference between what a refined atom and a base type atom need is exactly the information stored in the model information record as described above. All of the base type information records have another information pointer which can point to a model information record, or NIL if the instance is a base type. By designing it this way, base types only use the memory they need; and they can be refined by creating a model information record and linking it to the base type information record.

This change had drastic effects on both the running speed and the memory usage. The time needed to instantiate an instance was reduced by a factor of about two, and the memory usage by a factor of about ten.

2.2.8.2 Boolean

In addition to the model information pointer, a boolean information record contains a value attribute and an assign counter. The value of a boolean instance is true or false. The assign counter indicates how many times the instance has been assigned. This is used during instantiation and its use is described below. Figure 2-11 shows an example of an unrefined boolean information record.

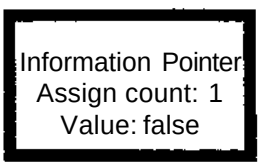

Figure 2-11: Boolean Information Record

2.2.8.3 Real

The real information pointer contains an assign counter, value, dimensions, equation list, and variable pointer. The assign counter has the same use as described in the boolean section above. The value is the value of the instance in system units which can be inferred from the dimensions specification. The equation list contains a list of all the relation information records where this instance appears in the expressions that form the relation. The equation list is used when ARE_THE_SAME'ing or deleting atoms (see Section 5.2.4). The variable pointer is a pointer to the solver's representation of this variable. Figure 2-12 shows an example of a real information record of a refined real.

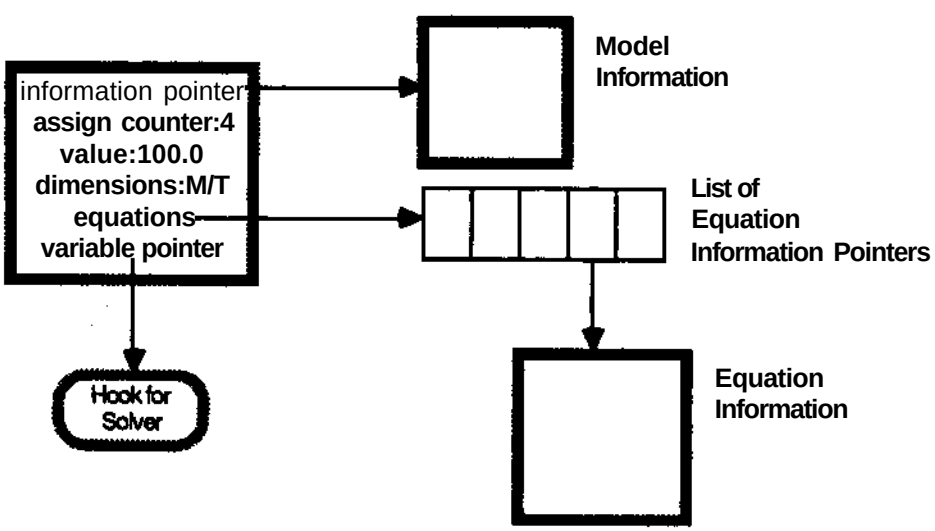

Figure 2-12: Real Information Record

2.2.8.4 Unit

The unit type exists because the model builder needs a way to specify the units in which a variable should be displayed. The unit information record contains an assign counter, unit name, dimensions, and scale factor. The assign counter is used as described above. The name is an unlimited string that presents the text of the unit (i.e. "psia" or "kg-moles/second"). The dimensions indicate the dimension of this unit. The scale factor is a number used to convert from internal units to these units. The scale factor is defined to be scale factor = <number in internal units>/<unit in these units>.

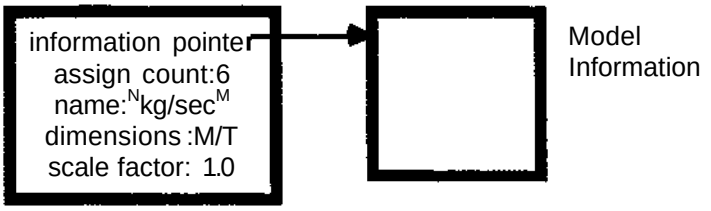

Figure 2-13: Unit Information Record

2.2.8.5 Integer

Integers have only two attributes - the value, and the fixed flag, (not to be confused with the fixed child of generic_real type). The fixed flag becomes true when a value is assigned to the integer instance. Once the fixed flag becomes true the value of the integer cannot be changed.

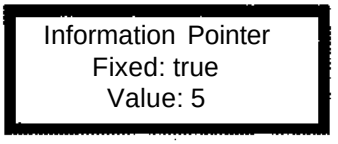

Figure 2-14: Integer Information Record

2.2.8.6 String

Strings have two attributes - value and fixed flag. The fixed flag has the same meaning as the fixed flag within an integer. The value of a string is an array of characters (fixed length). A fixed length string rather than an unlimited length string was used because of memory savings and because we did not have a flexible string handling module completed earlier enough. In the tests we have run, the limitation of 10 characters has not been a problem; however, certain applications (such as an extensive physical properties data bank) may require that the fixed length string be replace with an unlimited length string.

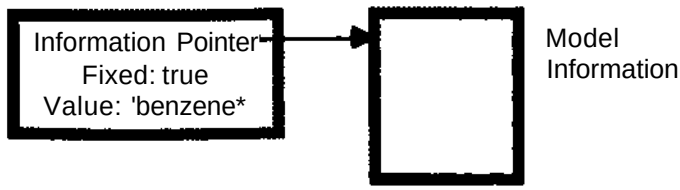

Figure 2-15: String Information Record

2.2.9 Relations

Relations are treated like objects or instances for flexibility and because they have a child (the "included" attribute). In discussing the language definition we considered treating relations as objects which could be declared ARE_THE_SAME or perhaps ARE_ALIKE. We eventually rejected that idea, but the implementation had to be flexible enough to handle relations as objects. The fact that relations have the boolean instance included as a child, makes it advantageous to treat the relation as an object. Otherwise, relations would have to be treated as a special case which would require a lot of additional. - code and increased instantiation time. Early versions of the Ascend interpreter did not treat relations as instances, and because of that could not provide all the needed functionality.

The attributes contained in a relation information record are the left hand side, right hand side, relation type, residual, included pointer, and equation pointer. The left hand side is the expression that appears to the left of the relation symbol (<, <=, =, >=, or >), and the right hand side is the expression that appears to the right of the relation symbol. These expressions are stored in postfix (also known as reverse Polish notation). The expression (4+3)/14+5 is "4 3 + 14 / 5 +" in postfix. Postfix is a very convenient way of storing expressions because the terms can be store in a list rather than in a tree. Left hand side and right hand side are lists of terms in postfix. A term can be any of the following: a pointer to an instance of base type real, a number, a binary mathematical operator (+, -, *, /), or a unary mathematical operator (-, sin, cos,...). The relation type indicates what kind of constraint the relation is <, <=, =, >=, or >. The included pointer points to the child boolean instance called included. Lastly, the equation pointer is a pointer to the solver's representation of this relation.

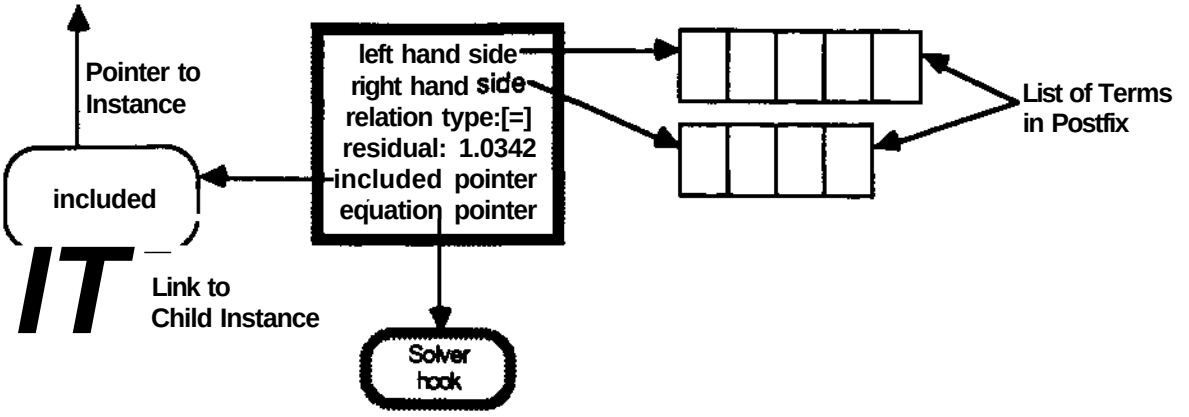

Figure 2-16: Relation Information Record

3. General Purpose Modules

3.1 List

Because lists are extensively used by the Ascend interpreter, the decision was made to write a single module to handle all occurrences, except the circular linked list used to represent cliques. The list module contains a set of procedures and functions for creating and manipulating lists. Improved performance could have been obtained by writing a different module for each occurrence; however, the increase in performance did not merit the additional code, programming, and debugging time.

The list module was designed specifically to meet the requirements of the interpreter. It was important to make the lists fast and memory efficient. For these reasons there is no error checking; It is assumed that the clients of the module will use it correctly. The module was designed to maximize the efficiency of the following operations which are extensively used by the interpreter: the addition of items to the front" and back of a list, and the sequential access of elements from top to bottom rather than random access. The former suggests a linked list with head and tail pointers which can add items to the bottom and top of the list very rapidly. In general, linked lists are not efficient for random access (access by position in the list); however, this is not a requirement for this application. To aid in the sequential access of the list, a current position pointer was added to the list data structure. The list module provides routines to do the following:

- create a list
- destroy a list
- retrieve the current list item (the one pointed to by the current position pointer)
- move the current position pointer
- delete the current list item
- sort the list based on a comparison function provided by the user of the list module

Figure 3-1 shows a graphic representation of the data structure used by the list module.

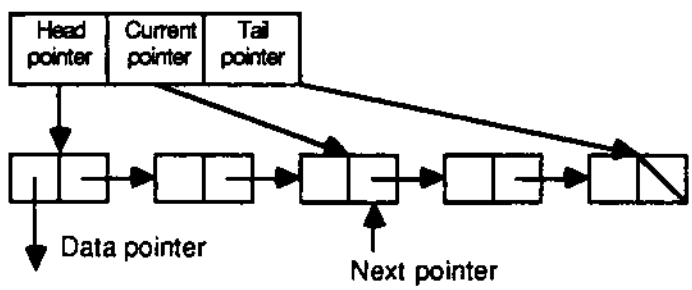

Figure 3-1: Representation of a List

3.2 Hash Table

Several parts of the interpreter need to reference items in a table by name, specifically the type library and children of atoms and models. In general the data structure that provides the fastest lookup by name is a hash table. A hash table has a fixed number of buckets (shown below Figure 3-2) and a hash function associated with it. Each bucket contains the list of items that belong to that bucket. These lists use the general list module described above. The hash function takes a string of characters and determines into which bucket the name should be stored. An example of a simple hashing function would be to look only at the first letter of the character string. Assuming you had 26 buckets, all the A's would be in bucket one, B's would be in bucket two, etc... The hash function used in this implementation is called hashpjw which is recommended in [1]. It uses all the characters to determine which bucket a character string would be stored in.

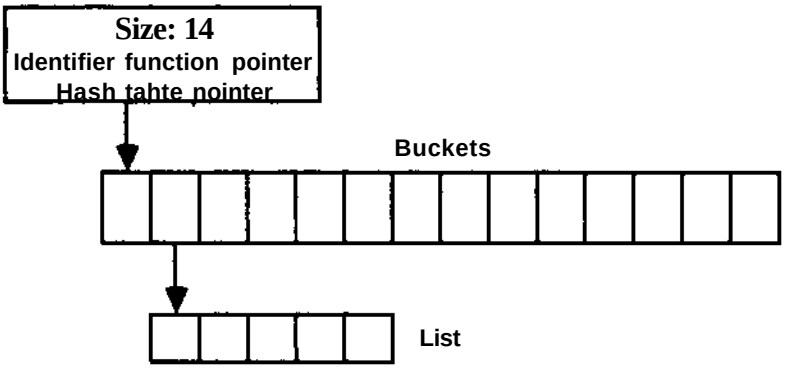

Figure 3-2: Representation of a Hash Table

The hash table module was not flexible enough to store children of arrays because children of arrays are indexed by integers or strings which cannot be incorporated easily into the hash function. The hash function, and supporting functions and procedures were designed to work for an array of 30 characters maxiumum. The hash function could be changed to work on integers and strings, but this would require substantial changes in the routines in the hash module. As mentioned above on page 8 the timing tests indicated this change would not dramatically improve the run time.

4. Parsing

From the start of **the** Ascend implementation, we planned to use a compiler-compiler tool to create the code to parse **the language.** There are **two** reasons **for** using a compiler-compiler rather than writing the parsing code ourselves. First, a compiler-compiler offers the flexibility to easily make changes to the language syntax. Second, it would have taken a considerable length of time to learn about, choose, and implement a sophisticated parsing algorithm. The ability to flexibly support changes to the language syntax was a key consideration.

The code to parse the description file is created by a pair of compiler tools called aardvark and llama [2] [5]. These two tools convert an extended Backus-Naur Form(EBNF) description of the language syntax into a Pascal module which can recognize and process files containing descriptions written in the Ascend language. From a functionality standpoint, aardvark and llama are Pascal equivalents of lex and yacc which produce C code. The purpose of the parsing code is to create the type library from the type definitions found in the description file. In addition, the parsing code will report syntactical errors found in the description file.

5. Instantiation

Instantiation is the process of converting a type definition into an instance tree. Once the instance tree has been created it can be interrogated, variables can be assigned, and the associated system of equations and variables can be solved.

Because Ascend is a declarative rather than procedural language, the execution of statements is order independent except for assignments; assignments must be executed in the order they appear in the type description. Complete creation of an instance requires that all of its defining statements must be simultaneously satisfied.

The instantiation procedures have been the most difficult part of the interpreter to write. Most of the difficulties are a consequence of flexible arrays, the prescence of which may result in instructions that cannot be executed in the order they occur in the user's type description. The following is an example of this situation which occurs often in Ascend descriptions.

```
model exaxnple_xnodel;
      a[integer] IS_A generic_real;
      n_iteras IS_A integer;
      a[l. .n_items] ARE_ALIKE;
      n_items := 10;
end examplejmodel;
```

Notice how the value of njtems is not available when the statement a[1 ..njtems] occurs. In this example, it would be easy to reorder the statements so that the problem would not occur, but in general this is not the case. The writing of general types(models and atoms) often leaves the values of the integer variables unspecified; they are later specified by the user of the model. We have developed the notion of *"pending instructions"* to cope with situations in which not all instructions can be executed. Integer and string instances are very important to the compiler because they are used to index arrays and because of this, a type cannot be fully instantiated until all the integer and string instances have received their values through an assignment or by an ARE_THE_SAME.

5.1 The Fixed Pass Instantiator - A Method That Failed

This section describes a method for instantiation that could not be guaranteed to work for all Ascend descriptions.

5.1.1 The Idea

The idea was to build the instance tree in a fixed number of passes, each pass having a specific task. Instructions that were not executed by a pass were stored in the instance tree until a subsequent pass could process them. A pass is defined as visiting each node of the current instance tree and executing some of the instructions in the pending instruction list for that node. The first several passes would execute statements that did not contain array references, the intent being that all integer and string instances would be assigned values; thereby totally specifying the instance tree. The final passes would then create array elements and complete the instantiation. This approach required that all array elements had to be identical, which in itself became a valid reason to reject the approach. Array processing was done by creating and refining a single prototype of the array elements during the initial passes. In the latter passes, array elements would be made by copying the prototype.

This idea was implemented and tested. It worked for several examples; but unfortunately, it did not work in general.

5.1.2 Why It Does Not Work

I believe that a fixed pass method will not instantiate all descriptions written using the Ascend language. The following example illustrates why the number of passes cannot be fixed.

```
modal another_example;
        il[string] IS_A integer;
        si[integer] IS_A string;
        i2[string] IS_A integer;
        s2[integer] IS_A string;

        il[sl[i2[s2[l]]]] := 1;
        sl[i2[s2[l]]] := 'benzene';
        i2[s2[l]] := 2;
        s2[l] := 'ridiculous';
end another_exaxnple;
```

Instantiation of the above requires a minimum of four passes because of the order of value assignments(:=). Each assignment cannot be executed until the one below it has been executed. Pass one executes ^{H}s2[1] := 'ridiculous';"; pass two executes ^{M}i2[s2[1]]:« 2;"; pass three executes ^{M}si[i2[s2[i]]] := 'benzene^{1};^{1}; and pass four executes ^{M}i1[s1[i2[s2[1]]]]:« 1 ;^{M}. Now if this model is extended with i3 and s3 up to i^{n} and s^{n} (i.e. i1[s1[i2[s2[...i^{n}[s^{n}[), it would require a minimum of 2xn passes to instantiate unless there was a way for the interpreter to rearrange the order of statements. For this example it is easy to suggest such a reordering; however, in general this not the case.

5.2 The Multi-Pass Instantiator - A Method That Works

5.2.1 Description

From the experience with the first interpreter, it was clear that the interpreter should be able to make an arbitrary number of passes through the instance tree. There seems to be no general way to process integers and strings before creating arrays, so this method makes no attempt to do so and does not require array elements be of the same type.

Before the multi-pass method can be described, the concept of a *partial instance* must be explained. Because the interpreter cannot always process the instructions in the order they were written (as demonstrated above), there must be a way to store the unexecuted instructions. In this implementation, the unexecuted statements are stored in the instance data structure (see Section 2.2.7 which shows where they are stored in the data structure). An instance which contains unexecuted statements is catfed a partial instance, and an instance tree that contains partial instances is called a partial instance tree. A partial instance or instance tree is an equivalent representation of the completely instantiated instance, because it contains all the relevant information from the type definition encoded either in the form of unexecuted statements or in the data structure itself (instances merged, cliques formed, etc.).

The multi-pass interpreter maintains a global list that contains all partial instances. It takes the instance on the top of the list and tries to execute its pending instructions. Executed instructions are removed from the pending instruction list. If the interpreter executes all the instructions, the instance is removed from the list; otherwise, it is moved to the back of the list. The above is repeated until the list is empty (all the statements have been resolved) or until it goes through the entire list without being able to execute an instruction (because the model is incorrectly defined). If the list is not empty, the list of partial instances is reported to the user.

For a more detailed description of the multi-pass method, it is best to look at an example of instantiation. This example will describe the process used to instance a type A.

1. The multi-pass method starts by creating a partial instance of A which contains all the statements from the definition of A. At this point, no statements have been executed.
2. A list is created to hold all the partial instances (instances with unexecuted statements). Initially, this list contains the partial instance of A which was created in 1. This list is called the pending instance list.
3. This step checks for loop termination.
    - If the pending instance list is empty, STOP (instantiation's complete).

- If this loop has gone through the whole list without executing an instruction, **STOP,** reporting an error (the instance is incorrectly defined).

Otherwise, continue the steps below.

4. Take the instance from the top of the pending instance list (called the working instance), and try to execute the statements in it.
5. If the execution of a statement results in the creation of a new instance, that instance is added to the top of the pending instance list (on top of the working instance). GOTO 3
6. If all the statements in the working instance were executed then remove the working instance from the pending instance list and GOTO 3.
7. Otherwise move the working instance from the top of the pending instance list to the bottom. GOTO 3

Step number 5 causes the interpreter to attempt a depth first instantiation. Depth first instantiation means that whenever an instance creates a child instance (a child is created by IS_A statements or by array references) the interpreter will try to fully instance the child before resuming instantiation of the parent. The reason I say *attempt* a depth first instantiation is because the child instance may not be fully instantiatable without some information provided by its parents. Depth first instantiation seemed to be the natural choice because it attempts to fully execute IS_A statements which create children. In descriptions where all integer and string variables are assigned values before their use, the type can be instantiated in one pass. A few runtime tests indicated that depth first instantiation worked faster.

5.2.2 Making Refinements

Several of the language constructs (ARE_ALIKE, ARE_THE_SAME, and IS_REFINED_TO), cause instances to be refined from their current type, C, to a more refined type, M. Since inheritance is purely additive in the Ascend language, this is fairly easy to do. Refinements can only add statements; and in the case of atoms, refinements can also change the default value or specify the dimensions. To refine an instance, the interpreter accumulates all the statements required to make the refinement, which is done by tracing the type ancestry of M back to C, and copying the statements found in the refinements. These accumulated statements are added to the instances pending statement list, and the instance is added to the pending instance list, because it now has statements which are not executed.

5.2.3 Making Cliques

The ARE_ALIKE and ARE_THE_SAME statements create cliques which are collections of instances which must always have the same type. If one instance in a clique is refined to a more refined type, all the instances in the clique are refined to the same type.

The implementation of ARE_THE_SAME (describe below in Section 5.2.4) merges several instances into one instance which automatically forces the instances to have the same type, so no special data structure is required to maintain cliques formed by ARE_THE_SAME statements. Cliques formed by ARE_ALIKE are implemented using a circular linked list.

The execution of an ARE_ALIKE statement which produces a clique is not very difficult. The instances to be made ALIKE are processed two at a time. To explain this process we consider a case of two instances, A and B which are type **ajype** and **bjype** respectively, **ajype** and **b_type** are checked for conformability by searching the ancestry of **ajype** looking for **b_type** and searching the ancestry of **bjype** looking for **ajype** . For example, if the ancestry of **ajype** contains **b_type,** then **ajype** and **bjype** are conformable; and **ajype** is more refined than **bjype.** Once it has been determined which instance has the more refined type, the less refined instance and all the instances that are in a clique with it are refined to the more refined type in the manner described above. Then the two cliques are merged into one clique. The merging may create a circular linked list (Figure 5-1), add an instance to a circular linked list (Figure 5-2), or merge two circular linked lists (Figure 5-3).

Figure 5-1: Creating a Clique From Two Instances

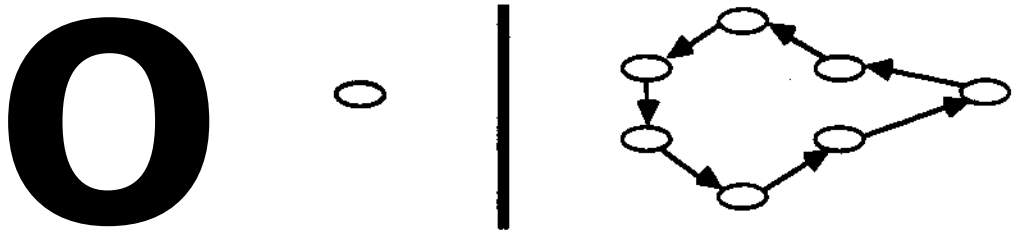

Figure 5-2: Add an Instance to a Clique

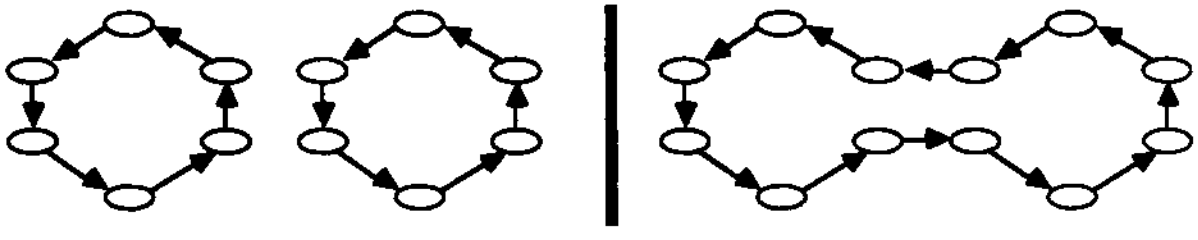

Figure 5-3: Merging Two Cliques

5.2.4 Comments on ARE_THE_SAME

The ARE_THE_SAME construct was the most difficult to implement. The reasons for this are the complex pointer manipulation and the management associated with partial instance trees. The management of partial instance trees is a subject worthy of discussion.

The problem can be reduced to the question of how to merge two partial instance trees.[3] One alternative is to only ARE_THE_SAME fully instantiated instance trees. This alternative has two disadvantages: it is slow because you have to check all the instances in both instance trees which could both be sizeable, and also there exists the potential for executing the same instruction twice. For example, if the two instance trees share the same unexecuted instructions, performing the ARE_THE_SAME would cut the number of instructions in half.

The way to merge two instances that have unexecuted instructions is to keep only the intersection of the two instruction lists. All the instructions that are not in the intersection have already been executed in one of the instances, and the effect of the instruction will be maintained in the merging process. The need to intersect statement lists requires that every statement be given a unique number in a consistent manner (meaning all instances of the same type have their statements numbered identically).

The merging process is difficult to explain, but I will try to present the gist of it. As for the ARE_ALIKE construct, instances (let's call them A and B) are merged two at a time. A will be the one which will be kept, and B will be destroyed. First, A and B are refined to the same type by declaring them ARE_ALIKE. All the instances that refer (point) to B are changed to refer to A (this includes parents and relations). If A and B are atoms, the value and dimensions are processed to obtain the merged value. The pending instruction list of A is replaced with the intersection of A's pending instruction list with the pending instruction list of B. B's instruction list is then destroyed. The children that occur in B and not in A are changed to be children of A. The children that occur in both A and B are recursively made ARE__THE_SAME. Finally whatever remains of B is destroyed.

The equation list is used when ARE_THE_SAME'ing two instances of base type real. These instances may appear in relations. Relations uses pointers to the instances of base type real. When ARE_THE_SAME'ing two reals, these pointers must be changed to point to the new merged real

---

[3] **All ARE_THE_SAME's are handled by merging two instances at a time. For example, A, B, C ARE_THE_SAME is actually executed A, B ARE__THE_SAME and then B, C ARE_THE_SAME. This can be done since ARE_THE_SAME is transitive**

instance. The equation list contains all the relations which refer to the real instance which therefore must be changed when the ARE_THE_SAME is executed.

5.2.5 Comments on Error Checking

The ability to detect errors in Ascend type descriptions during instantiation is limited, because the statements cannot be executed in the order they are written. Consider the statement "A.B, C ARE_THE_SAME;". If the interpreter searches for A.B and cannot find it, this means that either A does not have a child B (an error) or the statement which creates the child B has not been executed (not an error). It cannot distinguish between the error condition and non-error condition, so it cannot report errors as it evaluates statements. The current method of error reporting is as follows. If a type definition contains incorrect statements, when the multi-pass interpreter terminates it will return a non-empty pending instance list. The pending instance list will contain all the instances that contain statements that could not be executed. Any incorrect statements will be contained in the instances in the pending instance list.

6. Concluding Remarks

6.1 **Evaluation**

Size of the Ascend Interpreter	
Lines of Pascal Code[4]	14,884
Lines of EBNF[5]	**589**
Lines of Pascal Parsing Code[6]	3,029
Total Lines of Pascal Code	17,913
Not including code for the solver or graphic interface	

Some Example Timing Trials			
Number of Instances	Number of Equations	Time to Instance (CPU sec)	Model Description
364	32	4.334	integrate x using trapezoidal rule
574	52	6.375	integrate x^2 using Simpson's rule
929	73	18.722	simple simulation of a two phase stream being throttled and flashed
9,826	919	81.865	a fourth order Runga-Kutta simulation of the heat and material balance ODE's for a plug flow reactor [4]
All time trials were performed on an Apollo DN4000 with 20MB RAM running SR10			

Currently, the speed of the interpreter is somewhat disappointing. As the table above suggests even small simulations take a significant amount of time to instantiate. It is still useful for testing and demonstrating the functionality of Ascend. The writing of this prototype interpreter has already prompted changes which I believe will improve the instantiation speed.

[4]**This number includes blank lines and comments. It does not include the parsing code produced by aardvark and llama**

description file for aardvark and llama

[6]**Produced by aardvark and llama**

6.2 The Case Statement

The case statement has not been implemented yet, primarily because an acceptable method for doing so has not been found. There are many problems to be resolved in the implementation of a general case statement. For example, different cases can add instances and mathematical relations therby altering the degrees of freedom associated with the problem. In another example, drastic changes in the structure of the instance tree may occur through ARE_ALIKE and ARE_THE_SAME relations.

Because ARE_THE_SAME and ARE_ALIKE operators cause irreversible changes to the instance tree, a first approach to implementing the case statement might be to build a complete instance tree for each case within the case statement. For a type containing two case statements each having three cases, nine(3*3) complete instance trees would need to be created. In general this method is unacceptable; however, one could imagine an approach which minimized the redundancy by creating a single instance tree in which all structural alternatives were embedded.

6.3 Possible Improvements

In all the simulations we have tried, over 60 percent of the instances in the simulation's instance tree are fields of genericjreal. This is not suprising because each generic_real has 4 boolean attributes and 3 real attributes. These attributes of generic real are treated the same way as every other instance. However, since an atom can only contain instances of base types, atoms could be treated as one unit (one block of memory) rather than as an instance tree. As a result of this, atoms could be instanced and ARE_THE_SAME'ed much more rapidly. It would also reduce the memory required for atoms, which is important, since runtime analysis indicates that over 38% of instantiation time is spent allocating and deallocating memory. Developing a specialized memory management system would also increase the speed of instanciation.

Using a special format for library modules is another way of increasing instantiation time. This idea was mentioned briefly in Section 2.1. It is possible to design a method for storing partial instance trees in a library file. Whenever a statement instances a type that is in the library file, the interpreter can copy the instance in the library file rather than starting the instantiation from scratch. I believe that this idea has the potential to drastically reduce the instantiation time if most of the types are stored in library files.

Lastly, this interpreter could be adapted to work as an interactive interpreter processing each statement as it is entered. This is viable because the multi-pass interpreter can process partial instance trees. An

interactive interpreter would be even slower than the current interpreter and it would not be able to provide very good and immediate error reporting due to the problems mentioned in Section 5.2.5.

References

[1] Ahoetal.
CompilersPrinciples,TechniquesandTools.
Addison-Wesley, Reading, Mass., 1985.

[2] Brian J. Dunn and Tim J. Murphy.
Llama-ACompilerGenerator.
Technical Report, Department of Computer Science University of New South Wales, Sydney, Australia, 1976.

[3] Piela, Peter.
ASCEND:Anobject-orientedcomputerenvironmentformodelingandanalysis.
PhD thesis, Carnegie-Mellon University, December, 1988.
Publication pending.

[4] Smith, J. M.
ChemicalEngineeringKinetics.
McGrawHill, 1983.
Example 13.5.2.

[5] Bradley White.
Llama-Aardvark:ALexicalAnalyserGenerator.
Technical Report, Department of Computer Science University of New South Wales, Sydney, Australia, 1981.