The Layered Architecture of a System for Reasoning about Programs

Charles Rich

The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Cake is a hybrid system which provides reasoning facilities for the Programmer's Apprentice. This paper describes the architecture of Cake, which is divided into eight layers, each with associated representations and reasoning procedures. The operation of Cake is illustrated by a complete trace of the solution of an example reasoning problem. We also argue that a hybrid system in general is characterized by the use of multiple representations in the sense of multiple data abstractions, which does not necessarily imply distinct implementation data structures.

1. introduction

An earlier paper [12] describes the philosophy and overall design of a hybrid reasoning system with two levels — a bottom level of general purpose predicate calculus facilities, and a top level which supports a specialized planning language. This paper describes the implementation of this system (which has come to be called Cake) in more detail, with emphasis on the predicate calculus level. Two new results reported here are the refinement of the predicate calculus level into five separate layers, and a more subtle view of hybrid reasoning systems in general. We argue in this paper that a hybrid system is characterized by the use of multiple representations in the sense of multiple data abstractions, which does not necessarily (as is often interpreted) imply distinct implementation data structures.

The body of this paper is divided into m*ee* sections. The first section introduces a example program reasoning problem, discusses why program reasoning is hard in general, and points out the advantages we hope to gain with the current architecture. The next section describes the implementation of each layer of Cake in detail. Following this, we return to the example reasoning problem and trace through a complete scenario of how this problem is solved by the current Cake implementation. We conclude with a discussion of some difficulties with Cake and its relation to hybrid reasoning systems in general.

This report describes research done at the Artificial IntelNoence Laboratory of the Massachusetts Institute of Technology Support for the laboratory's artlfioial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grant MCS-8117633, and in part by the IBM Corporation.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor Implied, of the Department of Defense, of the National Science Foundation, nor of me IBM Corporation.

2. System Motivations

An Example Problem

Cake is being developed as the central knowledge representation and reasoning engine for the Programmer's Apprentice [8,9,14]. As such, its reasoning capabilities will be used to support analysis, synthesis and verification of programs.

Most of the knowledge in the Programmer's Apprentice is represented using a programming language independent formalism, called the Plan Calculus. A *plan* in this formalism is essentially a labelled, directed graph in which nodes represent program operations and data structures, and arcs represent the flow of data and control. A very important kind of reasoning involving plans which Cake must perform is the following: Given a plan and a set of correspondences between the incoming and outgoing arcs of the plan and the inputs and outputs of a given operation, determine whether the plan is a correct implementation of that operation (i.e. whether the plan satisfies the input-output specifications of the operation).

As an example of this kind of reasoning, suppose operations X, Y and Z are defined with the inputs, outputs, preconditions and postconditions shown in Figure 1, where the function g is commutative, the domain of the function f is A, the range of f is B, and B is a subtype of C.

Our problem is then to determine whether the plan XY, composed of operations X and Y with the data flow shown in Figure 2, is a correct implementation of the operation Z with the indicated input and output correspondences.

The solution to this problem is achieved by a kind of symbolic plan evaluation [13] roughly as follows: assume the preconditions of Z; using the correspondences between the inputs of Z and the incoming arcs of XY, prove the preconditions of X; assert the postconditions of X; using the data flow between X and Y, prove the preconditions of Y; assert the postconditions of Y; using the correspondences between the outputs of Z and outgoing arcs of XY, prove the postconditions of Z. If all of the proofs above are successful, then XY is a correct implementation of Z.

Why Is This Hard?

The example above is highly simplified and abstracted so that the complete trace of its execution can fit into a paper of this length. In particular, the size and total number of formulae involved in the proofs is tiny, and only relevant information is included in the problem statement (e.g. there are no unused postconditions).

If the Programmer's Apprentice were concerned only with problems of this size, the simplest approach would be to dump all

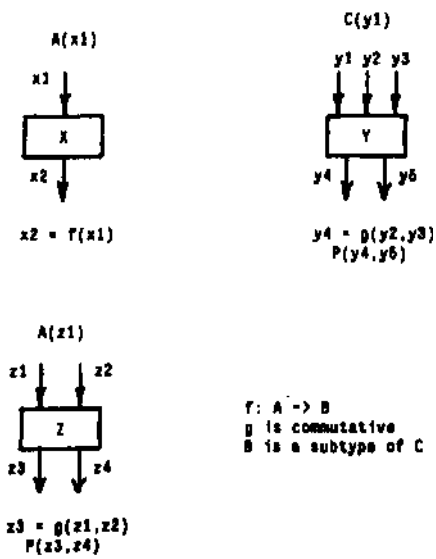
the axioms into a theorem prover with a single uniform general method (such as resolution), and wait for the result. The difficulty with this approach is that reasoning about programs of realistic size and complexity in this homogeneous way strains such systems beyond reasonable time and space limits, basically due to the inability to control the exploration of irrelevant proof paths. Furthermore, because we want the Programmer's Apprentice to support evolutionary program design, we require the reasoning system to support incremental retraction (i.e. truth maintenance), which existing uniform theorem provers do not.

The approach we are exploring in Cake is to partition the overall reasoning task among a number of specialized reasoning components. Even in the small example above, we can identify several distinct categories of reasoning which can be attacked with specialized algorithms: symbolic evaluation (of plans), equality (input/output correspondences), algebraic properties of operators (commutativity of g), functionality (f), and type inheritance (B is a subtype of C). As we will see below, the architecture of Cake has separate layers for each of these kinds of reasoning. What we hope to gain by this partitioning is the reduction to a controllable size of the reasoning problem seen by each layer.

3. The Architecture of Cake

The earlier paper on Cake [12] argued for the utility of partitioning the program reasoning task into two components, a plan level and and predicate calculus level. In this paper we carry this partitioning further. Figure 3 shows the architecture of the current Cake system, which is divided into eight layers. The bottom five layers (Truth Maintenance through Types) are the refinement of the predicate calculus level described in the earlier paper; the top three layers correspond to the plan level. The currently implemented Cake system includes only the bottom five layers plus the Plan Calculus (i.e. *not* the Plan Synthesis or Plan Recognition layers). We will restrict ourselves in the discussion below and in the example following primarily to the implemented portion of the system.

Fig. 1. Specification of Operations X, Y and Z



Before describing the principal facilities of each layer, we need to say a word about what is meant by "layer" in this context. Most concretely, each layer is a collection of subroutines and associated data structures. More abstractly, each layer adds a coherent increment of functionality to the system (i.e. the principal facilities summarized in the figure). The particular choice of layers shown here arose out of a mixture of bottom-up and top-down concerns. Partly we were motivated to take advantage of existing efficient algorithms for certain specialized kinds of reasoning (e.g. congruence closure), and partly we followed our intuition about parts of the overall reasoning task (e.g. type inheritance), in which the control problem could be attacked with only "local" context. Finally, we simply needed some kind of engineering discipline to help with the implementation of a large and complex system.

The formal modularity restriction between layers is quite weak: the subroutines in each layer may invoke only subroutines In the same layer or in layers below. This is not as strict as the conventional software engineering notion of a layered architecture, in which each layer may use only the layer immediately below. We retreated from this stricter notion for the usual reason, i.e. because it prevented many necessary optimizations. We will point out some of these inter-layer relationships in the sections which follow.

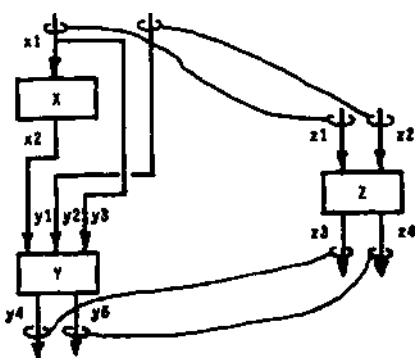
Before proceeding further in the description of Cake, we must gratefully acknowledge the use (with minor extensions) of McAllester's Reasoning Utility Package (RUP) [7] as the basis for the bottom three layers of Cake. One major extension which was made to RUP is noted below.

The Truth Maintenance Layer

The bottommost layer of Cake, Truth Maintenance, is essentially the boolean constraint propagation network from RUP. In this network a set of prepositional axioms is stored in disjunctive clausal form. The literals which make up the clauses ("constraints") are called *tms-nodes,* and each is assigned a value of true, false, or unknown. Each clause also has a documentation string for use in generating explanations.

The Truth Maintenance layer provides two principal facilities. The first is to act as a recording medium for dependencies, and thus to support retraction and explanation. The second is to perform simple "one-step" deductions (specifically, unit propositional resolution) based on the constraint clauses. This layer is at the bottom of the architecture because it provides an "active database" in which the final results of inferencing in all other layers must eventually be recorded. The data base also

Fig. 2. Implementation of Z as XY



performs simple inferences and limited contradiction detection. When a contradiction is detected in the network, control is passed to a user-provided routine to decide what to do (e.g. what premise to retract).

For example, if a reasoning process in a higher layer wishes to assert statement P supported by statements 0 and R, it obtains the tms-nodes corresponding to P, 0 and R, and installs the constraint clause,

P V - Q V - R ,

which is just the disjunctive normal form of 0 A R => P. The effect of installing this clause is twofold. First, it is available for answering the question "Why P?". The answer is "from Q and R by *...documentation string...".* One can then ask "Why Q?" and so on recursively. Second, the constraint propagation processing attached to the network will retract P whenever Q or R are retracted.

Given its role in the architecture of Cake, it is an important feature of the Truth Maintenance layer that deductive completeness has been traded off for control (McAllester [6] proves that the amount of computation triggered by a data base access is at worst linear in the size of the data base).

The Equality Layer

From the standpoint of the Truth Maintenance layer, tms-nodes are atomic — they have no internal structure other than their truth value. The Equality layer introduces the notion of *terms.* A term is defined recursively as either an atomic term (e.g. a symbol or a number), or a list of subterms (operator followed by arguments), each of which is a term. The two principal facilities supported by the Equality layer are the uniqueization of terms, and the incremental maintenance of a congruence (equality) relation on terms.

The uniqueization facility is a conventional hashing function which, given a list of terms, returns an existing term which has the given terms as subterms, or creates a new such term.

The incremental congruence closure facility operates roughly as follows (for a more complete description see [7]): Given any two terms, the Equality layer will tell you if they can be proved equal by (possibly recursive) substitution of equals using the set of currently true equalities between terms. Equalities are asserted and retracted like any other statement. Furthermore, the answer given by the Equality layer includes the list of currently true equalities which are used in the proof, so that the proper dependencies can be installed in the Truth Maintenance layer. Substitution of equals implemented in this way is a very powerful inference mechanism, and is used heavily by the other layers of Cake. Note that the congruence closure algorithm does not operate by creating all possible substitutions, since this is not only inefficient, but impossible in general, e.g. if f(x)= x.

There are two important connections between the Equality layer and the Truth Maintenance layer. First, the Equality layer uses the Truth Maintenance layer to keep track of the dependencies between substitutions it performs. Second, certain terms (those which are interpreted as boolean-valued) are associated with tms-nodes. The Equality layer guarantees that whenever two such terms are equated, appropriate clauses are installed in the constraint network to represent the logical equivalence between them.

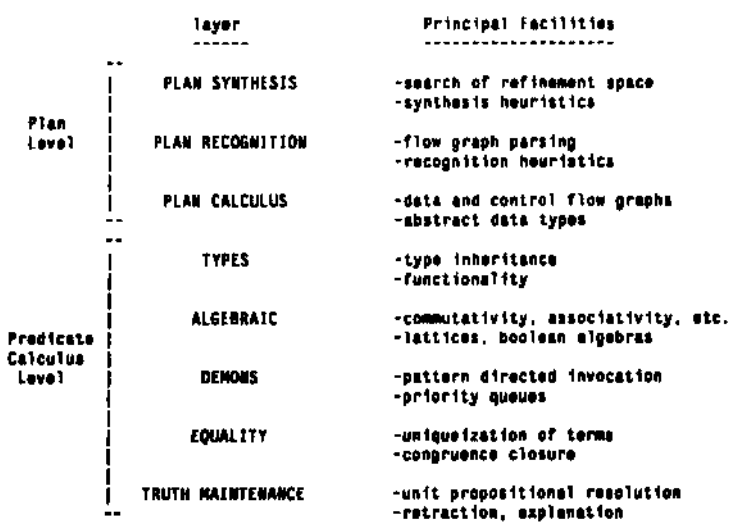
The Demon Layer

Demons are triggered by several different types of events in the Equality and Truth Maintenance layers. This layer of demons provides an interface through which many facilities in the layers above are implemented, and also a kind of "trap door" through which, with care, other miscellaneous kinds of inferences may be added to Cake. There are two important types of triggering event: the creation of a new term, and a change in the truth value of a tms-node.

Whenever a new term is created, the data base of demons is searched and all demons whose pattern matches the subterms of the new term are triggered. Patterns are currently one-level, with wild-cards but no variables. Whenever a new demon is added, it is triggered on all existing terms which match its pattern. The body of a demon is simply Lisp code which, when executed, may make new assertions (with dependencies), change truth values, or create new demons.

In a major extension to RUP, processing was added to new

Fig. 3. The Layers of Cake



term creation events to guarantee a kind of "completeness" in the interaction between the Equality layer and the Demon layer. In general, the Equality layer only creates a carefully controlled subset of all possible variant terms (terms obtained from other terms by substitution of equals). The added processing forces the creation of variants which would not otherwise be created by the Equality layer, if (and only if) such variants would cause a demon to be triggered. For example, suppose there is a demon with pattern "P(a,*)'\ and the term P(x,y) exists, and the equalities x = b and b = a are true. The completeness processing guarantees that the term P(a,y) is created.

Demons triggered by a change in the truth value of a tms-node have no patterns. Each demon of this type is attached to a specific tms-node. Both the old and new truth values can be made available to the body of the demon.

Examples of the use of demons will be seen in the description of the other layers and in the reasoning scenario.

The Algebraic Layer

The Algebraic layer is composed of special-purpose decision procedures for common algebraic properties of operators (functions and relations). The following nine properties may be specified to hold independently or in combination, either permanently or retractably, of any appropriate operator.

Reflexivity	Involution
Idempotency	Associativity
Commutativity	Distributivlty
Symmetry	Transitivity
Antisymmetry	

In addition to these nine basic properties, the Algebraic layer also has special procedures for reasoning in the following more complicated algebraic structures.

Partial Orders
Total Orders
Equivalence Relations
Lattices
Boolean Algebras

These properties and structures are useful for many different types of reasoning about programs, such as control flow (transitivity), arithmetic (distributivity of multiplication and addition), lists (associativity of append), and sets (boolean algebra). Although simplifiers and decision procedures for these kinds of properties are not new, Cake is the first such system which supports them in the context of retractable equalities (including equality between operators) and truth maintenance.

Most of the complexity of implementing these algebraic properties stems from attempting to control the creation of irrelevant terms and to guarantee completeness with respect to equality. To illustrate, consider commutativity, one of the simplest properties.

A commutative operator, f, obeys the law: $\forall x,y\ f(x,y) = f(y,x)$. To implement this law, we install a demon with the pattern "Commutative(*)". which is triggered by the assertion of a commutativity property. The creation of a term such as Commutative(f) triggers this demon, which then installs a new demon with the pattern "f(•,♦)". When triggered on a term such as f(x,y), this new demon checks whether the term f(y,x) exists; If It does, the demon installs the equality f(x,y)=f(y,x) with a dependency on the statement Commutative(f). If the term f(y,x) does not exist, a new demon is installed with the pattern "f(y,x)" and an empty body. The sole purpose of this third demon Is to force (via the "completeness" processing described above) the creation of the variant f(y,x) if it follows from other terms and equalities. The creation of this variant will in turn trigger the demon with pattern "f(•,♦)", which will install the equality f(v,x)=f(x,y).

For example, suppose the following statements are premises:

Commutative(f)
f = g
a = d
b - c

and we then enquire "Why g(a,b) = g(c,d) ?". Equality completeness processing for the pattern "Commutative(*)" forces creation of the term Commutative(g), which installs a demon with pattern "g(*,*)"- This demon triggers on g(a,b), installing an empty demon with pattern "g(b,a)". Completeness processing for this pattern forces creation of the term g(b,a) as a variant of g(c,d), which triggers the demon with pattern "g(*,*)" to install g(b,a) = g(a,b). The recursive explanation generated for this derivation is as follows:

g(a,b) = g(c,d) is true by transitivity of equality from:

1 ■ g(c,d) = g(b,a) which is true by subst'n of equals from:

1.1 b = c which is true as a premise.

1.2 a = d which is true as a premise.

2. g(b,a) = g(a,b) which is true by defn of commutativity from:

2.1 Commutative(g) which is true by subst'n of equals from:

2.1.1 Commutative(f) which is true as a premise.

2.1.2 f = g which is true as a premise.

The implementation of the more complicated algebraic structures listed above partly uses of the demons for individual properties (e.g. partial order uses transitivity, antisymmetry and reflexivity), but also uses specialized graph representations (e.g. the Haas diagram for a lattice). These graph abstractions are currently implemented using only the term indexing structures already provided by the Equality layer. This is one example in Cake of the use of multiple representations without separate implementations. Another example is discussed in greater detail in the Plan Calculus layer.

The Types Layer

Since the notion of data types is ubiquitous in reasoning about programs, it is natural to use a typed logic in Cake. The Types layer provides two principal facilities: a type hierarchy, and domain/range inferences.

Types in this layer are simply sets. Type inheritance is implemented using the boolean algebra facilities provided by the Algebraic layer. We can define subtypes, intersection, union and complement types; the appropriate inferences involving elements of the tyoe are performed incrementally with dependencies. For

example, suppose P, 0, R and Intersection(Q,R) are defined types, and P is a subtype of 0. If we assert the following type information about x (note that types are syntactically predicates; thus "x is an element of type P" is written $P(x))_t$

P(x)

then Algebraic layer demons, created by the Types layer, will deduce the following

Intersection(Q,R)(x).

In a typed logic, every operator, f, has a domain type, D, and a range type, R, such that the following axioms hold:

$$\forall x\ D(x) \Rightarrow R(f(x))$$

$$\forall x\ \neg D(x) \Rightarrow \text{Undefined}(f(x))$$

and similarly for operators of higher arity. Many useful properties of programs can be deduced using only this Kind of information about operators.

In the implementation of the Types layer, the first axiom above is implemented by a demon for each operator with a pattern of the form "f(*)" or '**(•,•)", etc. The second domain/range axiom above is implemented by special mechanisms for handling undefined terms.

The Plan Calculus Layer

In an early paper [11] we showed how to specify the semantics of the Plan Calculus by mapping each feature of a plan into predicate calculus. For example, the semantics of a data flow arc is an equality between terms which represent the source and destination of the arc; the semantics of control flow is captured using a global partial order (to express temporal precedence) and an equivalence relation (to express conditional execution). Since preconditions and postconditions are already logical formulae in the Plan Calculus, the only additional semantics to be specified for them is that the preconditions of an operation imply the postconditions. Structured data types, such as records and lists, are represented in the Plan Calculus as clusters of nodes with a logical constraint (invariant) between them.

In a succeeding paper [12] we reported that the predicate calculus formalization was directly usable as the basis for an implementation in which predicate calculus and plans co-exist. In this paper we wish to make a more subtle point about the relationship between the Plan Calculus and its "meaning postulates".

What makes Cake a hybrid system is that the current configuration of the plan for a program is represented by a set of true statements in the Truth Maintenance layer, and that the Plan Calculus layer provides a set of access functions supporting the node and arc view of plans, which is used by the Plan Recognition and Plan Synthesis layers to implement their respective algorithms. Whether or not the Plan Calculus layer has its own data structures is strictly an implementation question. (In this, we are taking a functional view of knowledge representation as advocated by Brachman et ai. [1].) In the current Cake system, the Plan Calculus layer is in fact strictly an interface to the lower layers. Later in the tuning of the system, we expect to introduce into this layer some caching and additional indexing of terms. However, these are just conventional software engineering concerns regarding the tradeoff between store vs. recompute.

The interface provided by the Plan Calculus layer includes functions for querying and altering any feature of a plan. Each querying and altering function embodies the semantics of that feature in terms of the corresponding logical formulae. For example, the access function for enquiring whether there is data flow between two operations generates a query to the Equality layer asking whether the appropriate two terms are equal; the access function for installing a new data flow arc asserts a new equality. Plan modifications made using the Plan Calculus layer are thus seen as a matter of course by subsequent queries from any layer.

Furthermore, the changes in truth values resulting from a plan modification can trigger demons and cause reasoning to take place, for example revealing a contradiction. Such contradictions can be reflected back into the Plan Calculus layer (e.g. for explanation) by way of extra-logical annotations installed by the altering functions. For example, the function for installing data flow marks the corresponding equality statement as representing data flow.

Of course, not all true statements in the Truth Maintenance layer correspond to some feature of a plan. Some statements have to do with reasoning internal to the theories of particular data structures, such as sets, lists, integers, and so on. Reasoning involving these terms can indirectly lead to a change in the truth value of some plan feature, such as a data flow equality. The dependency processing facilities of the Truth Maintenance layer guarantee that this change is propagated to all other features which depend on it.

Contradictions may also occur involving both plan and non-plan terms. This implies that a significant degree of flexibility is needed to dispatch the handling of a contradiction to the appropriate layer. Although facilities exist in Cake for locally binding and unbinding contradiction handlers, use of this part of Cake's control structure has not yet been explored.

Before going on to the trace of reasoning in the next section, let us briefly project future developments in the remaining two layers of Figure 3. The Plan Recognition layer will use Brotsky's algorithm for parsing flow graphs [3] for the "structural" part of recognition, and the lower layers of Cake for verifying the logical conditions. The Plan Synthesis layer will initially be a straightforward interactive plan generator (using a library of plan refinement rules), and will later use a general purpose planner such as Chapman's [5]. The Plan Synthesis layer is above the Plan Recognition layer because (as has been argued elsewhere [10]) non-trivial program synthesis must make use of recognition.

4. A Trace of Reasoning

To further illustrate the interaction of the layers of Cake, let us return now to the reasoning example of Section 2 and trace the complete execution of the solution. The scenario described here runs in the current implementation of Cake on a Symbolics 3600 Lisp Machine in approximately 10 seconds. In addition to the example problem statement here being a significant simplification as discussed in Section 2, the trace below is also a summarization in that it does not follow all alternative proof paths and it omits the irrelevant terms and clauses that are produced even in this small example.

Recall that the problem we were given, summarized in Figure 2, is to determine whether the plan XY is a correct implementation

of the operation Z, under the given input-output correspondences. The control structure of the solution comes from the symbolic evaluation of the plan XY in the Plan Recognition layer, which is currently being simulated by an ad hoc procedure.

The data flow arcs of the plan XY in the Plan Calculus layer are implemented in the Truth Maintenance layer by the following true equalities:

1. x2 = y1
2. x1 = y3

The input-output specifications of operations X and Y in the Plan Calculus are implemented in the Truth Maintenance layer by the constraint clauses resulting from reduction of the following formulae to disjunctive normal form:

3. A(x1) ⇒ x2 = f(x1)
4. C(y1) ⇒ [y4 = g(y2,y3) ∧ P(y4,y5)]

The "data theory" underlying this example includes the following true statement,

5. Commutative(g)

as well as several demons with patterns indicated below. The code in the body of each demon is summarized below by the quantified fact which it implements. First, there is a demon in the Algebraic layer which implements the definition of commutativity (as described above) for the function g.

"g(*,*)" ∀ xy Commutative(g) ⇒ g(x,y) = g(y,x)

The domain and range type of the function f is implemented in part by the following demon in the Types layer.

"f(.)" ∀ x A(x) =* B(f(x))

And finally, the fact that B is a subtype of C is implemented in part by the following demon associated with the type lattice in the Types layer.

"B(*)" ∀ x B(x) ⇒ C(x)

The symbolic evaluation begins by assuming the preconditions of Z:

6. A(z1)

We also take as premises the correspondences shown in Figure 2 between the inputs and outputs of the plan XY and the inputs and outputs of operation Z, which are implemented by the following true equalities:

7. z1 = x1
8. z2 « y2
9. z3 • y4
10. z4 « y6

Now, in order to recognize XY as an instance of Z we need to establish the truth of the postconditions of Z:

11. z3 = g(z1,z2) ?
12. P(z3,z4) ?

First we symbolically evaluate the operation X. The precondition of X,

13. A(x1)

is true by substitution of the equality of line 7 into line 6. Then, by the constraint of line 3 we get the postcondition of X:

14. x2 - f(x1)

Next we need to establish the preconditions of operation Y. The creation of the term f(x1) in the postconditions of X triggers the domain/range demon with pattern "f(*)", which installs the constraint clause,

15. A(x1) => B(f(x1))

which, with line 13 gives us:

16. B(f(x1))

The creation of this term triggers the subtype noticer with pattern "B(*)", which installs the constraint clause,

17. B(f(x1)) => C(f(x1))

by which it follows with line 16 that

18. C(f(x1))

By the operation of the Equality layer, the precondition of Y follows from line 18 and the equalities of line 1 and 14:

19. C(y1)

With the constraint clause of line 4, this gives us the postconditions of Y:

20. y4 = g(y2,y3)
21. P(y4,y5)

By substitution of the equalities from lines 9 and 10 into line 21 we establish line 12, one of the postconditions of Z (OED).

The creation of the term g(y2,y3) in line 4 triggers the demon with pattern "g(*,*)", which installs a demon with pattern "g(y3,y2)" and an empty body. Via the "completeness" processing between the Equality and Demon layers (as illustrated in the commutativity example earlier) this forces substitution of the equalities of lines 2,7 and 8 into the term g(z1 ,z2) yielding:

22. g(z1,z2) = g(y3,y2)

The creation of the term g(y3,y2) then triggers the commutativity demon with pattern "g(*,*)" again, which now installs the following constraint:

23. Commutative(g) => g(y3,y2) = g(y2,y3)

From this constraint and line 5 it follows that

24. g(y2,y3) - g(y3,y2)

is true. Line 11, the other postcondition of Z, follows by transitivity of equality from lines 9,20,22 and 24 (OED).

This completes the example trace. Note that as a byproduct of this reasoning, all the appropriate dependencies have been installed to support explanation and incremental retraction.

5. Conclusions

The system presented here is still under development. Although we do find that the hybrid approach gives us leverage on the control problem, there are still difficulties. For one, the interaction of the layers (especially involving the Algebraic layer) is highly tuned. Each additional increment of functionality has been difficult to add without causing explosions of computation. Also, on the topic of explosions, we are still concerned about the inherently exponential computations implicit in using a full

boolean algebra of types. Although this has yet not turned out to be a problem in practice, we may eventually be forced into restricting the power of the type hierarchy as in Krypton [2].

On the topic of hybrid reasoning systems in general, we simply would like to point out that the layered architecture of Cake is just one small point in a large spectrum of possible designs. There are many other options for how knowledge is partitioned, and how the individual components communicate and share information. Some of these general design issues in hybrid reasoning systems are discussed in a paper by Brotsky and Rich [4].

6. References

[1] R.J. Brachman, R.E. Fikes, and H.J. Levesque, "KRYPTON: A Functional Approach to Knowledge Representation", *IEEE Computer Magazine, Special Issue on Knowledge Representation,* pp. 67-73, October, 1983.

[2] R.J. Brachman, H.J. Levesque, "The Tractability of Subsumption in Frame-Based Description Languages**, *Proc. of the Fourth National Conference on Artificial Intelligence,* Austin, Texas, August, 1984.

[3] D. Brotsky, "An Algorithm for Parsing Row Graphs", (M.S. Thesis), MIT/AI/TR-704, March, 1984.

[4] D. Brotsky and C. Rich, "Issues in the Design of Hybrid Knowledge Representation and Reasoning Systems**, Proc. of *Workshop on Theoretical Issues in Natural Language Understanding,* Halifax, Nova Scotia, May, 1985.

[5] D. Chapman, "Nonlinear Planning: A Rigorous Reconstruction", *Proc. of the 9th Int. Joint Conf. on Artificial Intelligence,* Los Angeles, CA, August, 1985.

[6] D.A. McAllester, "An Outlook on Truth Maintenance", MIT/AIM-551, August, 1980.

[7] D.A. McAllester, "Reasoning Utility Package User's Manual", MIT/AIM-667, April, 1982.

[8] C. Rich and H. Shrobe, "Initial Report on A Lisp Programmer's Apprentice", *IEEE Trans, on Software Eng.,* Vol. 4, No. 5, November, 1978.

[9] C. Rich, H.E. Shrobe, and R.C. Waters, "An Overview of the Programmer's Apprentice", *Proc. of 6th Int. Joint Conf. on Artificial Intelligence,* Tokyo, Japan, August, 1979.

[10] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), June, 1981.

[11] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence,* Vancouver, Canada, August, 1981.

[12] C. Rich, "Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too", *Proc. of Second National Conf. on Artificial Intelligence,* Pittsburgh, PA, August, 1982.

[13] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", (Ph.D. Thesis), MIT/AI/TR-503, April, 1979.

[14] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Trans, on Software Eng.,* Vol SE-8, No. 1, January 1982.